Core Foundations, Algorithms, and Language Design for Symbolic Computation in Physics

A thesis submitted in partial fulfilment
of the requirement for the degree
of
Doctor of Philosophy in Physics
by
Jason F. Harris

University of Canterbury
1999

... let us enjoy the present exciting transition era, where we can both enjoy the rich human heritage of the past, and at the same time witness the first crude harbingers of the marvelous computer-mathematics revolution of the late 21st century. Zeilberger in D. Stanton (ed.). *q-Series and Partitions*, Springer-Verlag, New York, 1989.

Abstract

This thesis presents three contributions to the field of symbolic computation, followed by their application to symbolic physics computations.

The first contribution is to interfacing systems. The *Notation* package, which is developed in this thesis, allows the entry and the creation of advanced notations in the *Mathematica* symbolic computation system. In particular, a complete and functioning notation for both Dirac's BraKet notation as well as a full tensorial notation, are given herein.

The second part of the thesis introduces a prototype based rule inheritance language paradigm that is applicable to certain advanced pattern matching rewrite rule language models. In particular, an implementation is presented for *Mathematica*. After detailing this language extension, it is adopted throughout the rest of the thesis.

Finally, the third major contribution is a highly efficient algorithm to canonicalize tensorial expressions. By an innovative technique this algorithm avoids the dummy index relabeling problem. Further algorithmic optimizations are then presented. The complete algorithm handles linear symmetries such as the Bianchi identities. It also fully accommodates partial derivatives as well as mixed index classes.

These advances in language and notations are extensively demonstrated on problems in quantum mechanics, angular momentum, general relativity, and quasi-spin. It is shown that the developments in this thesis lead to an extremely flexible, extensible, and powerful working environment for the expression and ensuing calculation of symbolic physics computations.

Acknowledgments

There are a truly huge number of people that I would like to thank for their help, understanding and support in the creation of this thesis. It is hard to assign a relative order to the people and their contribution so let me first start with the obvious.

First I would like to thank the staff at the physics department. My supervisor, Professor Geoff Stedman for being understanding and letting me delve into the topics presented in this thesis. Also my mentors Associate Professor Phil Butler (now pro-vice chancellor) and Dr. William Moreaue.

I would also like to thank my graduate student colleagues: Andrew Richards, Tom King, Siva Nageleswaren, John Pritchard, Ed Savege, Caroline Bunn, and others. To my other good friends in Christchurch: Arnaud Scandrett, Alex Murray, Kirsten Conner, Neil MacBeth, and the crowd at the CCC. Moreover, I would further like to thank all of my good friends scattered around New Zealand and overseas: Paul StGeorge, Simon Ferrari, Megan Heart, Greg Jones, Greg Sutton, Chris Mean, James Webster, Peter Cadogen, Gerard Bolger, Sandra Pain, Chris Power, and Kath Ching.

I would also like to thank the people at the Wolfram Research Institute when I was there on scholarship. Specifically: Stephen Wolfram, Michael Trott, Robby Villages, Lou D'Andria, Rolf Carlson, Tommy Issavitch, and all the others.

In addition I would like to thank the people at Risc-Linz who graciously hosted me when I was there on scholarship, these included Bruno Buchberger, Daniela Vasaru, Claudio Dupre, Ovidiu Podisor, and Michael Joswig. Further, I would like to thank the support staff at RISC: Betina, Romna, Karoly.

Yet most importantly of all, I would like to thank my parents — Jean and John Harris. I cannot speak highly enough of them, their contribution to my development and their emotional support. Not that I ever plan to have children, but if ever did I am sure I could never be as good a parent as they have both been to me.

Lastly I would like to thank my partner Angela Smith for her unfailing and constant dedication to me and support through this long, fun, exciting, and at times arduous, task.

I am sure after this thesis is finally bound, I will remember someone that I have forgotten to put here. To all of you that I have missed: thank you!

Thank you all, for your time, efforts, support, and friendship.

Table Of Contents

Chapter 1
Introduction

1.1 History, Background, and Goals

1.1.1 The Opening...

It is hard to say exactly when symbolic computation became a field of research. Yet, one can now say with certainty that it is one. This thesis forms a contribution to the field of symbolic computation and its application to problems in physics. Before we embark upon a discussion of the background and history of the field, let us broadly say what this thesis attempts to, and arguably succeeds in, achieving.

Often problems in physics and mathematics share a common background. If one studies many of the computer packages and extensions for physics problems, one can discern patterns of commonality. There are certain operations which are fundamental to these packages and should be combined into a foundational layer. This foundation occurs in (*i*) the high level interface, (*ii*) the language used to express problems, and (*iii*) the algorithms used to solve particular problems.

It is vitally important that our developments are of practical use, thus any work such as ours must be tied to specific or hard computations. Consequently, this thesis must be intimately related to the capabilities of a computer algebra system — in our case, *Mathematica*. Yet, the work herein has implications, to be discussed, for the broader field of symbolic computation and physics beyond the *Mathematica* language.

This thesis can be broken down into four main parts:

1. Enhancements to the high level interface of our working language.
2. Fundamental modifications to the underlying language paradigm and the expression of computations.
3. Applications of both the interface and language enhancements to problems in physics.
4. An efficient and general algorithm for the simplification of tensorial expressions, and applications of this algorithm.

To the author's knowledge, all of these contributions are original. This thesis describes these original developments. To a large degree, the topics covered are intertwined.

The field of symbolic computation as applied to physics is a meld of physics, mathematics, and computer science. Necessarily then, this thesis is inter-disciplinary in nature. It unites aspects of functional programming, language design, rewrite rules, interfacing, parsing, object-oriented programming, quantum mechanics, operator theory, relativistic quantum mechanics, combinatorics, permutation group theory, computational group theory, algorithm design, and general relativity.

Although more detail will be given later, the notational advances are a vitally integral and important part of our overall system. The original advances allow one to perform computations in the notations in which one standardly works. Even though the enhancements are based upon the *Mathematica* front end — which in its own right is revolutionary — to the author's knowledge, such expressibility was simply not present in systems before the innovative work of this thesis.

The fundamental language modifications presented in this thesis can be categorized, in part, as context sensitive non-local rules, and in part as a prototype-based rule rewriting system. These modifications result in a paradigm which shares traits with many other paradigms such as abstract data types and object-oriented programming. It should be stressed that many applications or works are labeled "object-oriented" at the present time since it is a readily applicable "buzzword" in almost any area of computer science. Yet our developments are novel, and to the best of the author's knowledge, there are no comparable language extensions or viewpoints. Thus I claim that my developments in this area are original.

The tying together of these developments leads to a highly expressible and flexible working paradigm for expressing problems in physics and indeed other areas. The examples we present motivate the need for a general and efficient algorithm for canonicalizing tensorial expressions. We also present an advanced problem involving quasi-spin, where through the systems developed, we correct mistakes in some previously published works and find a different, more standard formulation for the quasi-spin operator.

The simplification of tensorial expressions lies at the core of many problems in physics — such as general relativity, Dirac algebra, second-quantized operator calculations, and indeed other areas. Consequently, finding an efficient, widely applicable algorithm that is not based on problem specific heuristics is of great importance. Due to an original innovation, the algorithm we develop for canonicalizing tensorial expressions is highly efficient. To the best of the author's knowledge, it is orders of magnitude faster than comparable algorithms.

To a large degree the original work in this thesis has all been motivated by problems the author has observed or encountered in physics. Yet, as is common in mathematical physics, the answers to the questions raised can be elegantly addressed in more abstract settings. Still, in the end, once the developments have been discovered and described, their application is once again to concrete physics problems.

It is imperative that the readers of this thesis have had some degree of exposure to symbolic computation. By this I mean they have used *Mathematica*, Maple, Macsyma, MuPad, REDUCE,

AXIOM, or a similar system so that they are familiar at least with the concept that symbolic computation is computing with symbols. (At a stretch, this class could include computational abstract algebra programs such as GAP, Cayley and Magma.) I do not believe an exposure to writing programs, say in Matlab or Pascal, is sufficient. Symbolic computation is the process of trying to automate algebraic computations on a computer system. If the reader has not had this exposure, then I fear they will not understand even the basic motivation for symbolic computation, let alone its advanced applications in this thesis.

1.1.2 Computer Algebra: Origins to the Present

According to various authors, notably Winkler[337], some of the very early work in computer algebra started in 1953 at the Massachusetts Institute of Technology (MIT). Davenport[87], as well as Geddes[124], provide an historical overview of the development of computer algebra. Symbolic computation has been, and for the foreseeable future is, inexorably tied to the development of computer science. Symbolic computation is, of course, also inescapably tied to mathematics. However, the limitations of the very early systems had more to do with a lack of knowledge with respect to programming languages and environments than to a lack of mathematical knowledge. Indeed, it was not until the rise of languages in which one could "reasonably" create programs/systems for performing algebraic computations, that the field of computer algebra developed. Foremost of such programming languages was Lisp[226, 227], developed at MIT. LISP also heralded the start of the development of modern functional languages.

The early systems were limited in several ways [87]: (*i*) the computational power of machines in which the systems were implemented was not sufficient for the programs; (*ii*) the algorithms were too simplistic; (*iii*) the interfaces to the programs were extremely cryptic and hence only usable by experts; (*iv*) the systems were inflexible or set up for just one selected task.

The early systems were actually batch process driven. Interactive computer algebra really only started in the late 1960's. At the present day most systems, barring the more specialist applications, are interactive. This is a necessary part of working and "playing" with systems, equations, models, etc. "Playing with a problem", as any physicist knows, is vital to understanding concepts in physics.

Since the dawn of computer algebra in the 1960's, there have been a plethora of languages covering many different applications and systems. However, from the mid 1960's on to the late 1970's authors became increasingly focused on providing general systems. Yet, as Davenport points out, "One person's general is another person's specific system". We leave a detailed listing of many of the "general" and "specific" languages to the aforementioned overviews, and restrict ourselves to just the major systems.

In the late 1960's Hearn[156] started the development of REDUCE, an interactive LISP based system for physics calculations. REDUCE has since then made the transition to a general purpose computer algebra system. Also in the very early 1970's Moses & Martin developed MACSYMA at MIT. At the time MACSYMA was one of the most advanced computer algebra

systems. However, MIT MACSYMA bifurcated into many different variants, most of which are now "dead". Only Macsyma[217] is still under development and maintained.

In the mid 1970's several specialist systems for calculations in physics emerged. These were primarily concerned with general relativity and high energy physics. The notable system in general relativity was SHEEP[115] and its descendants [7, 165, 298]. Also in 1967 the first version of SCHOONSHIP[319], a package for high energy physics calculations, was developed by Veltman. Algorithmic development continued strongly throughout the 1970's and 1980's, and indeed continues today[87].

The early 1980's saw the introduction of Maple by Gonnet & Geddes [57, 58, 59, 152, 244] at the University of Waterloo as well as SMP[[70, 134] by Wolfram at Caltech. The core of each of these systems was written in C. However, whereas Maple was designed to be a procedural language, SMP used a new language model, that of rewrite rules. Also, SMP was written with physics calculations explicitly in mind. The 1980's also marked the beginning of the commercialization of symbolic algebra systems. In 1988 Wolfram released *Mathematica* [341] the successor to SMP. *Mathematica* featured an integrated environment for numeric and symbolic computation and continued the language model of pattern matching and rewrite rules.

The Scratchpad system [136, 137], which had its origins in the 1970's at IBM Thomas J. Watson Research Center, developed into the AXIOM[177] system in the 1980's. To date, AXIOM is the only major "strongly typed" computer algebra system. The foundational paradigm of AXIOM, is based on principles of category theory and abstract data types. Although these concepts have a strong appeal in terms of rigor, for various reasons the system has not flourished. By the end of the decade, various other notable systems had been released including Derive[302] by Stoutemyer & Rich, as well as the group theory system Cayley[43, 44] by Canon.

In the 1990's the group theory package GAP[121] by Neubüser at Aachen was introduced, as well as Magma[223], the successor to Cayley. This decade also saw the introduction of the high energy physics package FORM[320, 321, 322] by Vermaseren, which has mostly supplanted the SCHOONSHIP system. The 1990's have seen continued development of the major computer algebra systems, the algorithms underlying these systems, and vast improvements in the user interfaces, notably *Mathematica* 3.0 [342] and its successor *Mathematica* 4.0 [343]. As a notable alternative to the commercial algebra systems, this decade also saw the introduction of the public domain computer algebra system MuPad[118, 119]. Loosely, it is similar to Maple, but more advanced in various respects.

Unfortunately but not surprisingly, for all the advances made, no computer algebra system yet matches a human in terms of "generalizability". For a simple example of this, consider the Schrödinger equation. Typically, to get to the Pauli-Schrödinger equation, one just adds a term like $\sigma \cdot B$ to the equation and says "Oh, by the way, that is a Pauli matrix, so we are now working with spinors." However, from a rigorous mathematical viewpoint, we have completely changed the setting in which we are working, and consequently to some degree any symbolic implementation will have to likewise be changed. Humans are exceedingly well adapted to such instances of "context switching", but less well adapted to performing enormous repetitive tasks. The complete opposite is true of symbolic computation systems. In some isolated areas, generalizability is occurring[346], but by almost all predictions, it is still a long way off.

Hopefully, at some stage in the future, symbolic systems will attain the generalizability and flexibility which is natural to humans.

1.1.3 Advances in Computer Science

Only through developments in computer science has it been possible to develop systems for computer algebra. Symbolic computation uses many ideas from computer science, including modularity, scope, abstraction, typing, language design, and a raft of other higher level concepts. To a large degree, the foundation of a symbolic computation system is just a specialized programming language, where hopefully the permissible structures and commands allow the easy creation of programs and structures for computations.

Von Neumann was instrumental in the creation of the computer systems we know today, since most computers use the von Neumann architecture, or at least derivatives thereof. We will not mention any of this specific history, since there are numerous articles and books which cover the origins of transistors, computer chips, machines and computation. The following historical summary is extremely brief and covers an eclectic set of points relevant to the content of this thesis.

The early languages were machine code based. Comparatively, it is far more difficult to write machine code programs since the commands used are of a "low level", yet machine code programs are usually the fastest since the coder can "hand tool" the algorithm to the architecture of the target machine. Because of the difficulties of writing machine code, people wanted to use other languages to express programs or concepts, and hence programming languages were created for this purpose. Language design and other concerns were instrumental in the genesis of computer science.

Early computer programming languages tended to be imperative in nature. That is, they consisted of a sequence of steps which included some simple iteration and branching statements. Typical of such languages were Fortran, Algol, Pascal, C and Ada. In a now famous statement, Wirth wrote "algorithms + data structures = programs". Pascal is an example of a "statically typed" language. In the Pascal language "everything" has a type, and moreover, this type can be discerned at compile time. As we later discuss in Chapter 4, the issue of types is not clear in symbolic computation. In typical computer languages or applications, one deals with say booleans, integers, floats, arrays, files, and other similar data structures. In symbolic computation we deal with myriad structures that are innately coercible to other structures[243, 281, 282, 332]. No attempt is made to comment on this issue, and the debate is ongoing in the community.

"Modern" programming languages, including functional languages followed imperative languages. Whereas imperative programming is closely tied to the von Neumann model, *functional programming* is based on functions and mathematics. LISP is an example of a *functional programming language*. A main driving motivation for the development of modern functional languages has been to allow better expressiveness and specification[169]. As mentioned above, it was really only with the advent of functional languages that the field of

symbolic computation started. Henceforth in this subsection, we discuss concepts in modern computer science. Background details are provided in [18, 22, 89, 125, 169, 178, 264, 310].

The more modern languages which are related to our needs are the functional languages, specification languages, and *object-oriented languages*. Examples of such relevant functional languages are, (*i*) LISP, which was used in many early symbolic computation systems, and is manifestly untyped; (*ii*) ML[146, 237, 259, 314] and its variants, which are pattern matching, strict evaluating, polymorphically typed languages; and (*iii*) Haskell[22, 89, 310] and its variants, which are also pattern matching based, polymorphically typed languages, but adopt a "lazy" as opposed to a strict evaluation model. Both ML and Haskell are rewriting languages [90]. It should be noted that there is much debate in the community of what exactly entails a functional language.

Let us briefly mention the features of the main symbolic computation systems, expressed in terms of the concepts of computer science. AXIOM is the only strongly typed language. Maple and *Mathematica* are untyped or extremely weakly typed languages. AXIOM is compiled, whereas *Mathematica* and Maple are in part semi-compiled or cached. Maple has an older style imperative core, while *Mathematica* has a functional rewrite rule core. AXIOM has an *abstract data type/category theory* paradigm. *Mathematica* by default has an eager evaluation mechanism, but there are ways to change this. Maple has a semi-eager evaluation mechanism, but again there are ways to change this too.

It should be pointed out that the pattern matching rewriting model used by *Mathematica* is comparatively advanced. There exist some extensions, such as [175], which border on allowing the same degree of expressiveness in patterns as does *Mathematica*. It is an important and outstanding problem to find a language as expressible as *Mathematica* yet one which is at least dynamically or incrementally compilable. Despite the superiority of *Mathematica* on the above mentioned points, it would be disingenuous to say that *Mathematica* is the best at everything. Various languages have various strengths. There are many factors which need to be taken into account when comparing languages[125].

Another model which has direct relevance to symbolic computation is graph rewriting[77, 264]. Graph rewriting is concerned with transforming multiply referenced structures within the setting of a graph as opposed to transforming a single term in an expression. Since much of an expression is typically duplicated in large symbolic expressions, if we work in a graph based setting, there is the potential to reduce or eliminate redundant computation. Unfortunately, this usually requires that we work in a *referentially transparent way* [22, 89, 259, 264, 310].

We are also concerned with the notation of *"higher-order languages"*[22, 89, 259, 264, 310, 338]. In such a language we can create and manipulate functions, and sometimes even procedures, in the language itself. The majority of the modern functional languages are higher-order, at least with respect to functions. Pascal is an example of a language which is not "higher-order". *Mathematica* is definitely a higher order language. This is important to us since the language model which we introduce is fundamentally based on being able to use higher-order language mechanisms. Indeed, we use this to a degree that computer scientists might find frightening. For more details see Chapter 4.

The concept of object-oriented programming is also important to us, specifically prototype based object-oriented systems. For reference on these concepts, see [18, 24, 252]. For further details, again see Chapter 4.

Specification theory and formal methods also share some commonality with symbolic algebra systems. But, as one of the invited speakers at the ISSAC'96 conference stated: "Formal systems are trusted but not used, while computer algebra systems are used but not trusted." In the large, to the best of the author's knowledge, this has so far apparently remained the case. Thus, despite the large amount of research into formal systems and specifications, unfortunately no substantial or real change has yet occurred in the main computer algebra systems.

As we design extensions to our system, we must be peripherally, and at times directly, aware of the concepts mentioned in this subsection. Hoare[162], amongst others, expresses similar sentiments when describing the role of a language designer. For further background in the literature on the interaction of computer science and the design of symbolic computation systems, consult the many papers found in the various proceedings of the Design and Implementation of Symbolic Computation Systems (DISCO) symposia, for instance [48, 110, 238, 239].

Most computer languages are in some sense "Turing equivalent"[229]. Thus, one might argue that one computer language is just as good as another, since anything one can do in one language can theoretically be done in another. *In practice*, this may be true in some limited imperative style programs, but in regards to more sophisticated languages, this is patently untrue. Indeed, language development and expressibility lie at the heart of the whole field of computer science. The expressiveness of the language one uses to create the computations or calculations directly influences the comprehensibility of the calculations. This is an obvious but deep truth. If a system is designed to be widely used, then the computations must be comprehensible. Moreover, if the system is comprehensible, it speeds up the adoption of the system.

In summary, the language in which we express our problems and computations is critical to the development of such computations.

1.1.4 Physics, Symbolic Computation, and Goals

The need for symbolic computation is clearly obvious. There are a large number of application areas and symbolic computation has solved some problems which would have otherwise been intractable. Many texts enter into detailing such applications — for instance see [65, 66], 312]. Beyond this, we will not justify the need for symbolic computation.

From the aforementioned reviews, the need for symbolic computation in physics should be equally apparent. There are multitudes of problems that are either cumbersome to tackle "by hand", or in all practicality impossible to tackle "by hand".

Most of the well known packages for physics in symbolic computation systems fall into only 3 categories:

(*i*) High energy physics, which tackles problems in Dirac algebra, Feynman diagrams, evaluation of resulting integrals, etc. The notable systems for tackling such problems are SCHOONSHIP[308, 319] and its acknowledged successor FORM[320, 321, 322]. There are also many excellent packages which interface to general symbolic computation systems. Amongst many others, such systems are FeynCalc[230, 231], FormCalc, *xloops*[33], Tracer[174], HIP[168], and Dill[208]. The review of [143] gives an excellent overview of the field.

(*ii*) General relativity, where the systems tackle problems in tensor canonicalization, expression manipulation, component calculations, metric classifications, etc. The notable systems in this area were SHEEP[115] its descendants [7, 165, 298], but now are Cartan[303, 304], EXCALC[286, 287, 288], GRTensorII[248, 249], MathTensor[258], REDTEN[144, 145], and MapleTensor[188, 189], to name the major ones. Also see [7, 15, 52, 56, 79, 80, 192, 193, 218, 219, 220, 317, 347]. For reviews of algebraic computing in general relativity, see [29, 67, 82, 83, 116, 209, 210].

(*iii*) Groups, Lie groups and group representation theory. Notable systems are Schur[345] and Dimsym[294].

Technical Note: Even though all discussion so far has been restricted to symbolic computation, it is currently the case that numerical computation dominates computational physics — see Pang[257] or Koonin & Meredith[198].

Despite the large number of computer algebra programs for specific areas of physics, there are no packages that provide an adequate framework for handling a broad class of "middle ground" physics, that is, core physics applications including standard calculations in quantum mechanics, angular momenta, operator expansions, quantum electrodynamics, etc.

Technical Note: Of course some specific perturbative expansion calculations in say quantum electrodynamics are expertly performed by systems such as FeynCalc amongst others, but other calculations are not.

One of the motivating examples for the packages and tools which are developed in this thesis was the problem of looking at higher-order terms in the non-relativistic limits of the Dirac equation. This involves some operators and their general handling and also some tensorial manipulations. Many problems in physics involve such mechanisms. Yet if one looks at the various packages in say *Mathematica*, such as FeynCalc[230], HIP[168], Dill[208], Tracer[174], Ricci[202], none are particularly suited, since they do not tackle this style of "problem" directly. NCAlgebra[160, 250] might be the closest such tool in *Mathematica*, but it is not well integrated into the underlying language. The REDUCE system would cover some aspects of such calculations, but again it leaves much to be desired in terms of the interface and extensibility of the underlying language. This is because the language is somewhat simplistic compared to modern languages for symbolic computation.

The problems above are not meant to criticize the aforementioned packages. Such packages are vital since the calculations in say high energy physics are truly large even by the scale of today's machines. Moreover, the perturbative calculations scale poorly, so the problems will not be totally alleviated by having faster machines. To achieve the large scale computations that are needed requires very efficient and specialized routines [143].

The middle part of this thesis presents something other than yet another package such as FeynCalc, HIP, Tracer, etc. Instead, it is concerned with the development of a general language framework which extends *Mathematica* in a direction in keeping with problem solving in physics. This general language extension is meant to not only allow the expression and manipulation of core physics problems, but it is also useful as a "foundation" on which one could write more specialized packages such as FeynCalc, HIP, Tracer, etc.

Our major concern is providing a language framework which will allow us to elegantly express physics problems. What language features are desirable? What will make our physics calculations readable? And equally as important what will make them writable? What structuring of the language facilitates the creation of reusable structures? These are classic central concerns of language design [125, 162, 340].

Earlier in my research, I created several programs for performing physics calculations in *Mathematica* that accomplished their specific goals and yielded answers. However, at this early stage, the calculations created were not even recognizable to the author after a few weeks. Indeed, it was hard to understand the programs which created the calculations since they looked nothing like the natural notations one normally uses on paper. From these early attempts to improve the working notations, I developed the notation package which now ships with *Mathematica*.

In addition to my early programs being hard to read, they were quite often closed in design, and insufficient in regards to generalizability. In essence, they were "throw away" programs. Possibly this may have been associated with my own lack of expertise, but I usually found the same "problems" with the code of others. Consequently, one of the foremost design goals was to avoid creating "just another package". I intended my system to be able to handle general problems in physics. I strongly desired a system which would have a "comprehensible", "modern", and "reusable" foundation. In essence I wanted to be able to perform calculations with the same clarity as I did on paper. These concerns prompted the creation of the language modifications of Chapter 4.

The applications of the notational and language changes are exhibited in Chapter 5 and Chapter 7. The calculations we perform in these sections are truly elegant in terms of their expressibility and readability. Moreover, they are comparatively efficient. One only has to examine some programs in say FORM or REDUCE to be convinced of the desperate need for the extensions such as those presented in this thesis.

The *Tensors* package grew from a need to be able to canonicalize tensorial expressions. The core algorithm is somewhat orthogonal to the language design concerns of the other parts of the thesis. In this sense the canonicalizing algorithm can be categorized as traditional applied "algorithm development" in symbolic computation. The algorithm we develop is orders of magnitude faster than other comparable algorithms. Yet, we then proceed to use our algorithm in conjunction with our language modifications to elegantly and cleanly present computations in physics.

This thesis consists of a dichotomy of complexity and simplicity. Our overall *raison d'être* for the work herein is to make computations "simple". Physicists or other scientists can often simply state some problem or other. Yet, upon "entering" such a problem into a symbolic computation

system, the implementation sometimes becomes unreadable to all but the implementor. When this is the case, the system has failed us. The primary reason for using a symbolic computation system is to compute results that were not easily possible on paper, such as when algebraic manipulations or expansions are too laborious. We have attained our design goals to the degree that our calculations seem intelligible, readable, understandable and extendable. Other systems in the main definitely do not share this combination of properties.

In summary, this thesis is an attempt to cleanly tie the areas of physics, symbolic computation, and computer science together to get a useful, expressive, structured, and extensible system for doing physics.

1.2 Overview and Conventions

1.2.1 The Choice

The Maple and *Mathematica* systems are now so large that thousands of man years of development have gone into each system. Of course it is inevitable that at some stage, once one has developed applications on any such system, one begins to encounter specific limitations, etc. One then faces a choice — both Hartley[150], and Davenport[87] also echo this. Does one maintain the use of the same underlying system? Or does one launch into the creation of a new system?

There is no universal answer, but as time progresses, the sensible answer is to use one of the pre-existing underlying systems. To attempt to recreate the monolithic systems with a small collection, or even a medium sized collection, of talented individuals is a herculean task. Maybe at some far flung future date the realities of the situation will change, and the proposals of the Open Math community[2] will be *de rigueur*, and every system will fluently communicate with every other system. But this date in all likelihood is far away.

Depending on the limitations of the problem, it may be possible to write a small to medium size program, say in C++, and then interface this with a pre-existing system. In this case, the problem is solved. However, some problems are not so easily tackled. If one does create a small stand-alone system, it is critically important that it can be easily linked to a system in which people standardly work. If not, then many tasks which will naturally arise in the course of any medium to large scale problem will have to be independently coded in the specialist system; or equivalently, one has to "reinvent the wheel".

Our choice was to work within the *Mathematica* system. However, as should be apparent from the coming chapters, we "extend" *Mathematica* to suit our needs. Indeed, if we had total flexibility, we would adopt much of the base of *Mathematica* but change the aspects with which we are concerned. Consequently, the work of this thesis is applicable beyond *Mathematica*. For instance, the *Notation* package or something similar should be a standard to which other

interfaces can attempt to match. Of course, since the other systems will have access to the fundamentals, they will be able to avoid much of the tricky manipulations we have to perform with box structures. Yet in essence, the ideas should be similar. For another instance, the canonicalization algorithm is manifestly applicable to canonicalizing tensorial expressions, independent of the calling system. Other than *Mathematica* being an ideal system in which to express the canonicalization algorithm, and allowing us to easily develop the algorithm, there are no intrinsic ties to *Mathematica*.

Yet despite the wider implications of our work, it is extremely important that we develop an elegant working system. This requires that we address many small details throughout the thesis that involve the language of our choice, namely *Mathematica*.

Geddes[124] notes "While all the languages of the sixties and seventies began as experiments, some of them were eventually put into 'production use' by scientists, engineers, and applied mathematicians outside of the original group of developers." It is extremely important that any language extensions *we develop* are not just experiments, but are actually useful to the wider community. Any package or system may be used by the authors of a system to compute specialized solutions to problems. Indeed, there is an extremely large number of such packages. However, to obtain the more elusive goal of contributing to the wider community, a package or system must satisfy one of the following conditions: (*i*) the system is an instrumental precursor to a new package or system which contributes to the wider community; or (*ii*) algorithms or routines which were previously unknown are developed in the package / system; or (*iii*) the package/system is cohesive and comprehensible enough to be used by the wider community.

The *Notation* package forms an original contribution to the wider community. This is corroborated by the fact that it is one of a select number of external packages which are shipping with *Mathematica*. There is also little doubt that the canonicalization algorithm forms an original contribution to the wider community since efficient algorithms to canonicalize tensorial expressions have long been sought [209] in general relativity packages and other packages, for the handling of tensorial expressions. Finally, there is no doubt that the language changes proposed are original. The physics calculations in this paradigm are extremely elegant and expressive. I hope and believe that they will also form a contribution to the wider field.

1.2.2 Superficial Overview

We now give a selective overview of the coming thesis. Each chapter has its own detailed introduction and conclusion, hence we defer specific introductions to the beginnings of the chapters themselves. However, the material of this thesis, after the present introductory chapter, can be roughly outlined as follows.

The first part deals with mathematical notations in computer algebra systems, as embodied in the *Notation* package developed by the author for *Mathematica*. Besides presenting the basic and more advanced features of the *Notation* package, Chapters 2 and 3 also contain two significant developments for the application of symbolic computing to physics. In Chapter 2 we present the first proper treatment of *functional* bra and ket vector notation in a symbolic system. And in Chapter 3 we present the first proper treatment, at least in *Mathematica* and probably in

any system, of *functional* tensorial notation in a symbolic system. Especially the latter is non-trivial and makes definite use of the advanced features of the *Notation* package. Heavy use of these notational advances are made throughout the rest of this thesis.

In Chapter 4 we develop the language extension to *Mathematica* that we use to express calculations in physics. In brief, it presents and provides examples of the prototype based, context sensitive, non-local, rewrite rule inheritance paradigm that we will adopt as our working paradigm.

The material in Chapters 2 through 4 may seem rather unrelated to physics for most physicists. However, without such a development, there is no way we could perform the symbolic physics calculations we do with the elegance that we do. Examples of such quantum mechanics calculations are given in Chapter 5. In that chapter, amongst other topics, after giving some basic generic prototypes, we use the example of the harmonic oscillator to perform calculations involving eigenbasis projection, normal ordering, creation and annihilation operators, computations with these operators, Hermitian conjugates, algebraic sums, the resolution of the identity, the propagator, and some time independent perturbation theory. We then treat angular momentum, calculations with primitive operators, raising and lowering operators, spherical harmonics, the addition of angular momenta, spherical tensor operators, and eigenket actions.

The third part of this thesis is devoted to tensors. In physics it is not uncommon to obtain expressions with hundreds of tensor terms, many of which are equivalent to each other under various symmetries. Thus, such complex expressions can potentially be greatly simplified. Various tensor packages available have routines for performing such reductions to a "canonical" form. They have various shortcomings, described elsewhere, one of the main ones being the lack of a good way to handle the "dummy index" problem. Chapter 6 provides the first proper solution of this problem and, in a definite sense, can be said to eliminate the problem. This allows the presentation of a "full" canonicalization routine for tensor expressions, more efficient than any previously developed. Actually Chapter 6 itself can be thought of as consisting of three parts. The first part is devoted to discussing and solving the dummy index problem and presenting the algorithm `BasicCanonicalize`. The second part develops the implementation and somewhat difficult mathematical theory behind the more advanced algorithm `OptimizedCanonicalize`. Finally, in the last part we take account of such advanced features as fixed index elevation arising through partial derivatives, mixed index classes, and linear symmetries. These combine to result in our "complete" algorithm `Canonicalize`.

The last chapter contains further applications demonstrating the language modifications, the notations, and the canonicalization algorithm, in general relativity as well as quasi-spin, and for illustrations sake some non-commutative dirac algebra.

1.2.3 Conventions Used

Often in the text we will speak of sets of rules or rule sets, or other types of sets. Usually in *Mathematica* these are implemented as lists. For the most part, we will use the terms 'set' and 'list' synonymously, and no confusion should be possible.

Each chapter starts in its own pristine *Mathematica* session. At the beginning of each chapter, the line numbering is reset to 0. Thus, we adopt the format followed by the *Mathematica* Book[342, 343] as well as various authors such as Trott[312] and Meader[222]. This is accomplished by the following code.

```
$Line = 0;
```

However, this resetting occurs inside "hidden input", a specific style set up in the thesis which does not appear in printing. Thus, this resetting is not explicitly shown. Inside each chapter any packages that are loaded or options set will remain loaded or set for the duration of that chapter.

In addition to the above, certain *Mathematica* defaults are changed. For instance, the output form of information is changed so that blank lines are not inserted, thus economizing space without causing any real detrimental loss of readability. Also, spell checking has been turned off for the duration of the examples in this thesis. This allows a more uncluttered explanation of the various examples and points. However, no package in practice turns spelling off. Thus, a user's experience may subtly vary from that portrayed in this thesis. But any variation should be extremely minor and recoverable by appropriately setting certain options. Moreover, executing any notebook chapter will result in the *exact* results printed here (modulo timings). There have been no "modifications" to output, and all results are "true and untampered with".

In terms of the programming style in *Mathematica* used throughout this thesis, the author has adopted more of a applicative style of notation like `f @ arg`, since this is more in keeping with modern functional languages such as Haskell[22, 310], Hugs[151], Clean[265], ML[146, 237, 259, 314] and its variants[CAML, SMLJ, MoscowM], logic languages, and even some symbolic computation languages like Aldor[31], which is the underlying language of AXIOM[177], etc. Thus, `f[g[x]]` will usually by written as `f @ g @ x`. This style of code, using @'s, is superbly effective when it comes to demystifying longer expressions. For a simple example, consider the utility function `padList`, which will be later used in various parts of the thesis. (`padList` pads a list by a given element.) We can code this function in two equivalent ways.

```
padList[list_List, padElement_] :=
   Drop[#, -1] & @ Flatten @ Thread @ {list, padElement};
padList[list_List, padElement_] :=
   Drop[Flatten[Thread[{list, padElement}]], -1];
```

In the author's experience, the first version appears to be more readable than the second. Indeed, generally the applicative style notation largely avoids the problem of denesting brackets, which plagues LISP[338]. Against the common style of *Mathematica* programs, we also adopt the applicative style notation. However, in some simple cases when there is only

one set of brackets, we will use the more conventional approach. In practice, the reader must be familiar with both approaches.

Another feature of the programming style in *Mathematica* adopted in this thesis is that pattern variables in expressions are almost always scripted. For instance, as we see in the above example coding of `padList`, both *ℓist* and *padℰlement* are scripted. This allows one to see which parts of an expression are pattern variables at a glance. The use of scripted variables is ubiquitous throughout this thesis.

Throughout the various chapters of this thesis some specific computations have associated timings. The base machine on which these timings are performed is a PowerMac 7600/4Gb/80Mb with a 300MHz G3 accelerator running in 2:1 mode with 1 Mb of backside Cache.

In this thesis the word *semantics* will refer to the meaning of certain operators or operations. For instance, a typical usage might be "... In the following subsection, we give the semantics of the tensorial assignment operator ...". By the semantics of an operator or operation, we are referring to its behavior and meaning, inside *Mathematica*, via a set of rules or definitions. Be aware that this definition differs from that used in formal programming semantics such as operational, denotational, or axiomatic semantics [125]. However, this usage appears to be common enough in computer science, and no real confusion should arise once this proviso has been noted.

Throughout the thesis, I use the terms or identifiers `foo`, `goo`, `bar`, etc., and a few others to be generic functions that are used to illustrate certain points. The functions have no special significance and are only used to illustrate a generic program, expression, or form. This is a common place practice in computer science texts.

This thesis attempts to provide major extensions to the basic *Mathematica* language that are useful to a wide range of disciplines, but most importantly physics. Many of the application areas are motivated by physics problems and concerns. However, describing and documenting these extensions is best done from a theoretical base. The situation is much the same as when physics books seem clearer, but may be less enlightening, because they adopt a pragmatic rather than an historical approach. Of course when we *apply* our extensions, we do so in a physics setting.

The physics notations in this thesis should be recognizable to any physicist; however, the notations may appear to be slightly different from normal. For instance, a bra-ket is denoted by $\langle \psi_f \mid \cdot \hat{\mathcal{H}} \cdot \mid \psi_i \rangle$, whereas in normal physics notation a bra-ket is denoted by $\langle \psi_f \mid \hat{\mathcal{H}} \mid \psi_i \rangle$. This divergence arises from the fact that we must be able to manipulate bra's and ket's in isolation from an overall bra-ket, thus for consistency, we must actually have an operator, in this case a '$\cdot$', present in the actual operator product. This is explained in greater detail in Chapter 5. Similar notational differences are true with creation and annihilation operators.

Chapter 2
The *Notation* Package

2.1 Introduction

2.1.1 Background

Throughout the evolution of *Computer Algebra Systems*(CAS), an extremely desirable feature and objective that has been strived for is a functional and elegant user interface. (For a survey of user interfaces, see Kajler & Soiffer[185]. For a history of notations see Cajori[47].) Undeniably, the easier it is to interact with a computer algebra system, the more it will be used. An important part of the user interface is the ability to enter expressions and display results in ways that are standard to each field of use. If a user has to contend with an awkward linear syntax or deal with an unfamiliar way of representing his or her standard ideas, the computer algebra system or package is less likely to be used. Correspondingly, the closer a user interface reproduces the standard representations of a field, the more likely it is to be used. Therefore, it is extremely important to be able to present computations in the notations specific to the field of use.

This chapter and its sequel describes the *Notation* package, how to use it, what it can accomplish, and what advances are expected in the future. The *Notation* package allows users to define their own general notations. When a notation is defined, it creates special behaviors for the interpretation of input and the formatting of output that involve the syntactical notation. The *Notation* package is an add-on package to *Mathematica* and is installed when *Mathematica* is installed. It first appeared in the *Mathematica* 3.0 distribution, which was released in 1996, and has been updated several times since then. The latest revision, at the time of writing, is the version being shipped with *Mathematica* 4.0. When *Mathematica* is mentioned, then unless stated otherwise, the reference is to *Mathematica* 4.0 or any later versions.

The remainder of this introduction presents some of the motivations for creating the *Notation* package. The beginning sections of this chapter describe how to use the package and how to create notations. The later sections discuss some of the advanced features of the *Notation* package, culminating in §2.7 *Example: Bra-Ket Notation* with an implementation of Dirac's bra-ket notation. In the following chapter we discuss, among other things, the principles underlying the *Notation* package, the creation of a notation for tensors, give further complex examples and outline the planned future enhancements.

A *notation* in our context involves the correspondence between the way a mathematician might represent a concept and the form that a computer might handle. For example, consider the expression $\int_0^\infty e^{-x}\,dx$ and its *Mathematica* ASCII counterpart `Integrate[ Exp[-x], {x, 0, Infinity} ]`, or the correspondence between $\forall\, x \ni |\, x - x_0\, | < \delta$ and `ForAll[x, Abs[x - x0] < δ]`. (In the last expression we did not exclusively use ASCII since δ is not a character in the ASCII language and the x_0 is subscripted.) These two examples illustrate that it is invariably easier to understand an expression if it is presented in a standard way.

There have been many advances in notational systems leading up to the modern computer algebra systems. To date, the major computer algebra systems have varying degrees of graphically typeset output and to a lesser extent allow some degree of graphically typeset input. These systems can be loosely broken down into three categories:

- Commercial systems, such as *Mathematica*[342, 343], Maple[57, 58, 59, 152, 244], MACSYMA[217], AXIOM[177], Derive [302], MathCad [225], and Theorist [25].
- Specialized and non-commercial systems, such as GAP [121], Cayley[43, 44], and its successor Magma [27, 50, 223], REDUCE [155], and MuPAD[119].
- Configurable shell systems, such as CAS/PI [183], CaminoReal [10, 11], and SUI[94] ; strictly, these are not computer algebra systems but interfaces to other computer algebra engines.

Generally, the commercial systems have the best interfaces, while some of the shell systems, particularly CAS/PI, also have progressive interfaces. However, the shell systems are usually not as "polished" or "integrated" due to their developers' lack of resources. The idea of having a shell system which is able to interface with different computational engines is an appealing one, and there is ongoing work on this idea, for example, the Open Math System [3, 85], and IZIC [112, 113].

In almost every major system, a common focus has been to have or develop a notational system that is recognizable. Bosma & Cannon *et al.*[26] state that one of their overriding design principles of Magma was that the system should have a notation as closely resembling recognized algebraic notation as possible. The Maple [57, 58, 59, 152, 244] system has moved progressively towards better typeset notations. The latest version of AXIOM (Jenks & Sutor 1992 [177]) has a system that interacts with a TeX[194] style environment. Many of the other systems such as MuPAD[119], MACSYMA[217], and Derive[302] have also moved towards better notational interfaces. Even some of the older systems like REDUCE now have some comparatively simple graphical extensions [49].

It is obvious that a computer algebra system which allows input and presents output in a "standard" and recognizable form has a clear advantage over a computer algebra system which requires the use of "non-standard" or non-recognizable input and output. Researchers will be more interested in results presented in the notationally superior computer algebra system, since they then do not have to translate these results from arcane linear syntactical expressions. They will approach such systems with more familiarity and greater understanding. In addition, the documents produced will also be easier to prepare for electronic publishing. Moreover, using a familiar notation can greatly demystify the coding of algorithms specific to a field. Therefore, the ability to easily represent objects can sometimes drive the development of mathematical algorithms dealing with these objects. In summary, the issue of notations is a critical one.

Some systems allow the user to introduce new operators and notations; for example, in Maple the user can override *neutral operators*. However, most systems are quite limited in how far one can add new notations. *Mathematica* lets the user modify the functions `MakeExpression` and `MakeBoxes`, which are at the heart of the parsing and formatting of notations; and by doing this, the user can add new notations. Unfortunately, without the *Notation* package, creating `MakeExpression` and `MakeBoxes` rules becomes problematic for anything but trivial additions.

Currently, the only two systems, apart from *Mathematica*, known to the author which allow the user to change the meaning of notations in a general way are CAS/PI[183] and MuPAD[119]. Unfortunately, in CAS/PI the only way to add new notations is through expertly defining new grammar rules in a separate language. For more complicated expressions, this becomes very difficult. The MuPAD group have very recently announced a new front end (Postel & Hillebrand[268]), produced by SciFace[289], which appears to have more potential to handle customized notations. It is not yet clear how well integrated and extensible their mechanisms are. Indeed, it is the author's understanding that one has to compile new templates in order to introduce new notations; but it may be that this is somehow internally automated when a MuPAD kernel is running. Certainly, the SciFace interface system postdates the introduction of the *Notation* package (1996), and in all likelihood has "borrowed" from the style and results of *Mathematica*. Thus, with the addition of the *Notation* package, *Mathematica* was, and probably still is, unique amongst computer algebra systems in its capabilities to add new notations that are usable in input as well as output.

The underlying structure of a two dimensional expression in *Mathematica*, such as $\int_0^\infty e^{-x}\,dx$, is defined in terms of *boxes*. These boxes represent the underlying structure of an expression. For instance, the expression $\int_0^\infty e^{-x}\,dx$ is represented by the box structure

```
RowBox[{SubsuperscriptBox["∫", "0", "∞"],
      RowBox[{ SuperscriptBox["e", RowBox[{"-", "x"}]],
            RowBox[{"d", "x"}]}]}]
```

For readability and usability, it is necessary to be able to introduce new notations easily, intuitively, and graphically. The *Notation* package achieves this functionality. Without the *Notation* package, it is necessary to define new notations by constructing explicit `MakeExpression` and `MakeBoxes` rules. With the *Notation* package loaded, one can use its three main utility functions — `Notation`, `Symbolize` and `InfixNotation` — to define new notations. At the internal level the package works as a compiler, translating `Notation`, `Symbolize` and `InfixNotation` statements into corresponding box structure manipulation rules, which are generally long and visually unintuitive.

If problems arise using the *Notation* package or when working through the electronic notebook, it is extremely helpful to check the expression structure of input and output. One can reveal the underlying structure of an expression by clicking in the cell that contains the expression and then using the Show Expression command from the Format menu of the front end.

Before attempting to use the *Notation* package, the reader must digest at least the next section, *Vital Preliminaries*.

2.1.2 Vital Preliminaries

In the introduction we have briefly given some motivation for why it is necessary to be able to define and use new notations. In the next two sections — §2.2 *Notation* and §2.3 *Symbolizations, Infix Notations, and InputAliases* — we will introduce the three main functions of the package: `Notation`, `Symbolize` and `InfixNotation`. These sections contain many examples and give the flavor of how the *Notation* package works. In the present subsection we will address some vital issues concerning conventions and the entering of notations that will be used throughout the rest of this thesis. In particular, we will briefly discuss (*i*) what we mean by a notation statement, (*ii*) the method for entering new notations, and (*iii*) the method for entering expressions which involve newly defined notations.

The *Notation* package is loaded in the usual way.

```
In[1]:= <<Utilities`Notation`
```

It is best that the reader wait to use the *Notation* package until after reading at least the next section. However, for those who will not wait, the following is essential information. To declare a new notation, it is *critically* important that one makes use of the notation palettes, shown in Figure 2.1.A below, containing the notation templates. (The first one is for *Mathematica* 3.0; the second, for *Mathematica* 4.0.) It is also *critically* important that, having declared a new notation, one does not try to copy and paste parts of it to form new input.

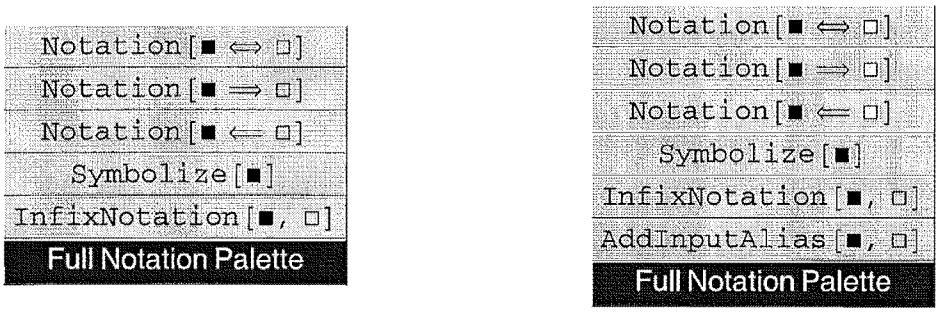


Figure 2.1.A

The notation palette is automatically loaded when the *Notation* package is loaded. If hidden, the palette may be obtained by selecting **NotationPalette.nb** from the **Window** menu. If accidently closed, it may be retrieved by opening the **NotationPalette.nb** notebook.

In *Mathematica* 4.0 one can often avoid the explicit use of the notation palette since each of the template buttons on this palette has its own pre-defined alias. That is, a `Notation` statement template can be created by ESC`notation`ESC, a `Symbolize` statement template can be created by ESC`symbolize`ESC, etc. Moreover, the notation palette contains a sixth template, `AddInputAlias[■,■]`, allowing users to introduce their own aliases for expressions, possibly but not necessarily containing notations introduced by `Notation`, `Symbolize` or `InfixNotation` statements.

The reason that the notation palette must be used when *declaring* a notation is that it pastes a template whose box structure contains essential hidden tag boxes embedded in the correct way (tag boxes will be scrutinized later, in §2.5.4 *Tag Boxes in Notation Statements* and §3.2.2 *Tag Boxes*). These embedded tag boxes allow `Notation`, `Symbolize` and `InfixNotation` statements to properly group and parse the new notations being defined. Furthermore, they allow these statements to capture the styling information of the new notation so that the output is formatted with the same spacing, sizes, adjustments, etc. as the input. Similarly, the reason that parts of a notation declaration must not be copied and then pasted to form new input is that the essential tag boxes contained in these parts of the declaration are detrimental to input. Both of these reasons will be explained in greater detail in later sections.

The conventional concept of a "notation" is that of a style, form or representation of syntactic marks. However, throughout this thesis the word 'notation' will generally refer to something *defined* or *created* by a `Notation` statement. Occasionally, the word 'notation' will refer to anything declared by a `Notation`, `Symbolize`, or `InfixNotation` statement, and on rare occasions can even refer to the conventional notion of a notation. The usage should be clear from the context.

By analogy, a *notation statement* is a statement produced by one of the three main notation creating functions of the *Notation* package, that is, either a `Notation` statement, a `Symbolize` statement or an `InfixNotation` statement.

In later sections, specific functions will be introduced enabling one to remove individual notations that have already been declared. However, if one would like to collectively remove all notations present inside the system, one can use the command `ClearNotations[]`. This command removes all notations, symbolizations, and infix notations active in the current *Mathematica* session.

`ClearNotations[]`	clears all notations, symbolizations, and infix notations

Syntax for clearing notations.

Technical Note: Using `ClearNotations[]` should not affect other packages that do not use the *Notation* package, nor should it affect definitions that have been made via `MakeExpression` or `MakeBoxes`.

2.2 Notation

In this section, we introduce and give examples of how to use the function `Notation`, the first of the three main functions provided by the *Notation* package for introducing new notations.

2.2.1 Notation: An Introduction

In our description of the `Notation` function, we will use the notions of *parsing* and *formatting* — see Aho[6], Bennett[20], Watson[330]. In the context of *Mathematica*, parsing is the transforming of input to a kernel expression and formatting is the reverse process of transforming a kernel expression to output. (See §2.5.2 *Parsing and Formatting* for an extended discussion.) For example, *Mathematica* standardly parses the external input `3 + x^2` to the internal expression `Plus[ 3, Power[ x, 2 ]]`, and conversely formats this internal expression as the external output $3 + x^2$.

The function `Notation` takes both an external and an internal representation as arguments. `Notation` forces *Mathematica* to translate (parse) any input matching the external representation into the corresponding internal representation, and/or it forces *Mathematica* to output any internal expression matching the internal representation in the form of the corresponding external representation. The various types of `Notation` statements are given in the following table.

`Notation[`*external* ⟺ *internal*`]`	set *external* to be parsed to *internal* and *internal* to be formatted as *external*
`Notation[`*external* ⟹ *internal*`]`	set *external* to be parsed to *internal*
`Notation[`*external* ⟸ *internal*`]`	set *internal* to be formatted as *external*
`Notation[`*notation*`, `*opts*`]`	define a notation *notation* with the options *opts*
`RemoveNotation[`*notation*`]`	remove the notation *notation*

Syntax of notation declarations.

Let us illustrate `Notation[ ... ⟺ ... ]` with a simple example. Assume we want to declare a notation for a new operator, to be represented externally by $\oplus$ with a subscripted parameter and internally by the function `gplus`. Currently, expressions of the form $x \oplus_\lambda y$ are not admissible.

```
In[1]:= x⊕λ y

        Syntax::sntxi : Incomplete expression; more input is needed.

        x⊕λ y
```

We can implement the desired notation effortlessly with the following declaration which, as explained in §2.1.2 *Vital Preliminaries*, we must enter using the `Notation[□ ⟺ □]` template. (Recall that one can obtain this template via either the *Notation* palette, or via the input alias ESC`notation`ESC.)

```
In[1]:= Notation[x_ ⊕λ_ y_ ⟺ gplus[x_, y_, λ_]]
```

The above notation having been declared, any input of the form $x \oplus_\lambda y$ is now admissible and is interpreted as `gplus[`x`, `y`, `λ`]`.

```
In[2]:= a ⊕α+β b // FullForm
Out[2]//FullForm=
        gplus[a, b, Plus[α, β]]
```

Conversely, any `gplus` expression is now formatted in the newly declared notation.

```
In[3]:= gplus[α, β, Γ]
Out[3]= α ⊕Γ β
```

The underscores in the `Notation` statement above represent patterns, just as in normal *Mathematica* rules. What is unusual is that the underscores appear on both sides of the notation statement. Let us defer a full explanation of this, and for now just say that they are necessary because the single statement sets up transformations in both directions.

Before we progress on to more involved examples, let us give one more simple example of a `Notation` statement. Let us remove a trivial irritation that arises in the everyday use of *Mathematica* when solving differential equations. In solutions, all constants are normally returned in the form `C[`n`]` rather than in the textbook form C_n.

```
In[4]:= DSolve[y''[x] == e^(i x) + y[x], y[x], x]
```

```
In[4]:= {{y[x] → -e^(i x)/2 + e^-x C[1] + e^x C[2]}}

In[5]:= Notation[C_n_ ⟺ C[n_]]

In[6]:= %%

Out[6]= {{y[x] → -e^(i x)/2 + e^-x C_1 + e^x C_2}}
```

2.2.2 Notation: Examples

The above notation for C_n solves a minor irritation in the everyday use of *Mathematica*. However, the main focus of the *Notation* package is to give the user the ability to define notations for input and output that would normally be inadmissible to *Mathematica*. The notation $x \oplus_\lambda y$ was one such example. In this subsection, we consider several more examples of notations, many of them practical, which would normally be inadmissible to *Mathematica*. Let us start by creating notations for the Laplace transform and its inverse.

```
In[7]:= Notation[ℒ_t_→s_[f_] ⟺ LaplaceTransform[f_, t_, s_]]
```

This new notation for the Laplace transform is now acceptable and is interpreted correctly.

```
In[8]:= ℒ_t→s[Sin[t]]

Out[8]= 1/(1 + s^2)
```

Applying the Laplace transform to a linear combination of general functions gives us the corresponding linear combination of Laplace transforms, because of the linearity property of the Laplace transform.

```
In[9]:= ℒ_t→s[a f[t] + b g[t]]

Out[9]= a ℒ_t→s[f[t]] + b ℒ_t→s[g[t]]
```

This confirms that our notation is working correctly with input and output. Further to our notation for the Laplace transform, let us create a notation for the inverse Laplace transform.

```
In[10]:= Notation[ℒ^-1_s_→t_[F_] ⟺ InverseLaplaceTransform[F_, s_, t_]]

In[11]:= ℒ^-1_s→t[1/(1 + s^2)]

Out[11]= Sin[t]

In[12]:= ℒ^-1_s→t[a F[s] + b G[s]]

Out[12]= ℒ^-1_s→t[a F[s] + b G[s]]
```

In *Mathematica* 4.0, the *Notation* palette contains a template button, `AddInputAlias[■, ■]`, which allows a user to add to the current notebook his or her own input alias for a specific notation. For example, we could introduce aliases for the above Laplace and inverse Laplace Transforms as follows.

```
In[13]:= AddInputAlias[ℒ■→■, "laplace"]

In[14]:= AddInputAlias[ℒ■→■^-1, "invlaplace"]
```

Subsequently, any time one wants the Laplace or the inverse Laplace transform template, one need only enter [ESC]laplace[ESC] or [ESC]invlaplace[ESC], respectively.

Technical Note: When an input alias is introduced, it applies only to the notebook in which it was entered. This is in contrast to the fact that notations, symbolizations, and infix notations work session-wide. Unfortunately, the limited scope of the input aliases is a consequence of the current front end technology. Hopefully, this will be rectified in later releases.

Next, let us proceed to consider an example which introduces a non-standard notation involving the integral symbol. For instance, let us define a notation for a hypothetical `genericIntegral`.

```
In[15]:= Notation[∫_𝒟_ f_ ⟺ genericIntegral[f_, 𝒟_]]

In[16]:= ∫_𝒜 δ' // FullForm

Out[16]//FullForm=
         genericIntegral[Derivative[1][δ], 𝒜]
```

Clearly, generic integrals are now acceptable in input. Similarly, assuming `genericIntegral` is itself undefined, any `genericIntegral` expression is now formatted in the newly declared notation.

```
In[17]:= genericIntegral[δ', Action[λ]]

Out[17]= ∫_Action[λ] δ'
```

Later, in §2.6.5 *Changing Grouping Behavior*, we will define a generic partial derivative notation to complement our generic integral notation. In addition, we will define some toy rules for the functioning of generic integrals and generic derivatives.

Our next example creates a notation for the Wigner 3-*j* symbol, which is used in the coupling of angular momenta in quantum mechanics — cf. Racah[272, 273, 274], Brink & Satchler[30], Rotenberg[277], Condon & Shortley[75]. The 3-*j* symbol is a more symmetric form of a corresponding Clebsch-Gordon coefficient.

```
In[18]:= Notation[ ( j1_  j2_  j3_ )
                   ( m1_  m2_  m3_ )_3j  ⟺
                   ThreeJSymbol[{j1_, m1_}, {j2_, m2_}, {j3_, m3_}] ]
```

This notation consists of a 3×2 grid box, with round braces subscripted by '3*j*'. Here is a typical calculation involving the Wigner 3-*j* symbol.

```
In[19]:= ( j      j + 1/2    1/2 )
         ( m    -m - 1/2     1/2 )3j

Out[19]= -(-1)^(-j+m) Sqrt[1 + j + m] / (Sqrt[1 + 2 j] Sqrt[2 + 2 j])
```

As before, we can add an input alias for this notation.

```
In[20]:= AddInputAlias[( □  □  □ )
                       ( □  □  □ )3j, "3j"]
```

A blank template of a Wigner 3-*j* symbol can now be created by typing ESC3jESC in any input cell. Until §2.3.5 *Adding Input Aliases*, we will not add any further input aliases for notations, symbolizations, or infix notations, although it is very easy to do so.

In standard *Mathematica*, abstract sums are only admissible between limits, if at all. The final example of this subsection addresses this shortcoming, by providing at least a notation for expressing more abstract sums. (In §5.4.8 *Algebraic Sums*, we will provide a natural semantics for this syntax.)

```
In[21]:= Notation[ Sum_indices__ expr_ ⟺ AlgebraicSum[expr_, indices__]]
```

As the following shows, this notation works correctly with both input and output.

```
In[22]:= Sum_{γ∈Γ, λ∈Λ} x_{γ,λ} // FullForm

Out[22]//FullForm=
AlgebraicSum[Subscript[x, γ, λ], γ∈Γ, λ∈Λ]

In[23]:= AlgebraicSum[x_γ + y_λ, γ∈Γ, λ∈Λ]

Out[23]= Sum_{γ∈Γ, λ∈Λ} (x_γ + y_λ)
```

Almost every character and operator in the *Mathematica* palette **CompleteCharacters** can be used inside a "notation statement". To reiterate, by a *notation statement* we mean a statement produced by one of the three main notation creating functions of the *Notation* package, that is, either a `Notation` statement, a `Symbolize` statement or an `InfixNotation` statement.

In the next subsection, we give some examples implementing a semantics for some of our customized notations.

2.2.3 Notation: Assignments

In this subsection, we demonstrate, as one expects, that our notations can be used almost everywhere in normal input. In particular, notations can be used inside assignment statements. This has the great benefit of allowing definitions in the language in which the author / coder "thinks". To illustrate, let us introduce some notations, and then proceed to define some semantics for the introduced notations.

For our first example, consider the Laplacian. The traditional notation for a Laplacian is normally unacceptable.

```
In[24]:= ∇² f_

         Syntax::sntxi : Incomplete expression; more input is needed.

         ∇² f_
```

Including brackets around `f_` does not help. With a `Notation` statement, however, we can make it acceptable.

```
In[24]:= Notation[ ∇²_coords__ f_ ⟺ Laplacian[f_, coords__] ]
```

We assign the properties of the 3-dimensional Cartesian and the 2-dimensional polar coordinate versions of the Laplacian to our `Laplacian` as follows.

```
In[25]:= ∇²_{x,y,z} f_ := ∂_{x,x} f + ∂_{y,y} f + ∂_{z,z} f

         ∇²_{r,θ} f_ := ∂_{r,r} f + 1/r ∂_r f + 1/r² ∂_{θ,θ} f
```

Observe that we have entered the definitions in terms of the newly created notation! That the above definitions function correctly is demonstrated by the following.

```
In[27]:= ∇²_{x,y,z} (x² y³ z⁴)

Out[27]= 12 x² y³ z² + 6 x² y z⁴ + 2 y³ z⁴

In[28]:= ∇²_{r,θ} (r⁴ Sin[2 θ])

Out[28]= 12 r² Sin[2 θ]
```

In a more abstract setting, the following declares a notation for arrows overscripted by `Apply`, which works with both input and output.

```
In[29]:= Notation[f_ --Apply--> expr_ ⟺ myApply[f_, expr_]]

In[30]:= u --Apply--> v // FullForm

Out[30]//FullForm=
         myApply[u, v]
```

```
In[31]:= myApply[u, v]

Out[31]= u ⟶(Apply) v
```

Simple rules like linearity can now be entered in a visually intuitive way.

```
In[32]:= a_ ⟶(Apply) (b_ + c_) := a ⟶(Apply) b + a ⟶(Apply) c

In[33]:= f ⟶(Apply) (Sin[x] + Log[x])

Out[33]= f ⟶(Apply) Log[x] + f ⟶(Apply) Sin[x]
```

For the third and final example of this subsection, let us next extend *Mathematica*'s ability to build lists with specific conditions. We use a `Notation` statement to introduce one list-building notation which mimics standard set-building notation, and then provide a semantics for the notation.

```
In[34]:= Notation[{ans_ | domain_, cond_} ⟺ ListOfAll[ans_, domain_, cond_]]
```

The following assigns the semantics for `ListOfAll`.

```
In[35]:= {f_ | x_Symbol ∈ list_, condition_} :=
           Function[x, f] /@ Select[list, Function[x, condition]];
         SetAttributes[ListOfAll, HoldAll];
```

Here are two simple illustrations of this notation.

```
In[37]:= {i^2 + 1 | i ∈ Range[20], PrimeQ[i]}

Out[37]= {5, 10, 26, 50, 122, 170, 290, 362}

In[38]:= {i → i^2 + 1 | i ∈ Range[10], Sin[π^5 i] > 0}

Out[38]= {2 → 5, 3 → 10, 6 → 37, 9 → 82, 10 → 101}
```

This kind of list constructor would be the precursor to more general Zermelo-Fraenkel set theoretic constructors, see Monk[245]. Actually, there are some shortcomings in the way we defined the above list-building notation. These will be discussed when we have more fully developed some of the necessary background. Specifically, this example is briefly revisited in §2.7.2 *Prototypical Ket Structures using Named Styles.*

The ability to use notations in definitions and not just calculations is of *extraordinary* benefit, as we shall see throughout the entirety of this thesis.

2.2.4 Notation: Explicit Bracketing

The *Notation* package has been expressly set up so that when a notation is introduced using a `Notation` statement, then what appears on the left-hand side, except spaces and simple patterns, is taken literally, that is, as part of the notation being introduced. Thus, apart from spaces and simple patterns, everything appearing in the notation statement must also be present in any input that is intended to use that notation. (The same holds true for `Symbolize` and `InfixNotation` statements, to be discussed in §2.3 *Symbolizations, Infix Notations, and InputAliases*.) For example, consider

```
In[39]:= Notation[(expr___)$ ⟺ Semantic[expr___]]
```

When using the above notation, $ is recognized as a `Semantic` wrapper if and only if parentheses are included in the input expression.

```
In[40]:= FullForm /@ {(a, b)$, H$}
Out[40]= {Semantic[a, b], Subscript[H, $]}
```

Thus, the brackets need to be literally present for the notation to function.

One could even use `Notation` to introduce high-level changes or additions to existing basic *Mathematica* notation. For example, some misguided individual might want to denote lists using angle-brackets '⟨ ⟩' rather than the standard curly brackets '{ }'. To accomplish this effortlessly, we enter the following.

```
In[41]:= Notation[⟨elms___⟩ ⟺ {elms___}]
```

The new notation is now active and works with both input and output, as expected and desired.

```
In[42]:= {a, 1, b, 2}
Out[42]= ⟨a, 1, b, 2⟩
In[43]:= MatrixForm /@ ⟨⟨1, 2⟩, ⟨3, 4⟩⟩, {{a, b}, {c, d}}⟩
```

$$\text{Out[43]=}\quad \left\langle \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \begin{pmatrix} a & b \\ c & d \end{pmatrix} \right\rangle$$

However, such a fundamental and radical change of standard *Mathematica* notation is to be greatly discouraged. Hence, we remove it immediately.

```
In[44]:= RemoveNotation[⟨elms___⟩ ⟺ {elms___}]
```

As we see in the above examples, each of the external and internal parts of a notation statement can have embedded patterns. However, there are added complications when the patterns are complex ones — for example, if they contain embedded conditions. So, until the next chapter, we will assume that all patterns inside notation statements are *simple patterns*, namely of the form *label_*, *label__*, or *label___*. Once a new notation involving simple patterns has been declared, one is free to use it like any other *Mathematica* notation. Therefore, of course one is free to use complex patterns in definitions that *use* the new notation.

We can put the forgoing together to create a new notation for functions, more in keeping with the way mathematicians denote pure functions.

```
In[45]:= Notation[ (arg_ ↦ body_)& ⇔ Function[arg_, body_]]
```

```
In[46]:= Function[x, x^2 + 1]
```

```
Out[46]= (x ↦ x^2 + 1)&
```

```
In[47]:= f = (x ↦ x^3 + Sin[2 x])&
```

```
Out[47]= (x ↦ x^3 + Sin[2 x])&
```

Clearly, our new notation is working correctly in both input and output. We can, of course, use these pure functions as one normally would.

```
In[48]:= f'
```

```
Out[48]= (x ↦ 3 x^2 + 2 Cos[2 x])&
```

Our notation also works for functions of more than one variable.

```
In[49]:= g = ({x, y} ↦ x^3 y^2)&
```

```
Out[49]= ({x, y} ↦ x^3 y^2)&
```

```
In[50]:= (g[x + h, y] - g[x, y]) / h
```

$$\text{Out[50]=}\ \frac{-x^3 y^2 + (h+x)^3 y^2}{h}$$

As a motivation for our next example, observe that the standard `Integrate` function in *Mathematica* can not handle pure functions.

```
In[51]:= ∫ f dx
```

```
Out[51]= x ( (x ↦ x^3 + Sin[2 x])& )
```

Unquestionably, this is mathematical nonsense. We could override `Integrate`, but instead let us introduce a notation and a definition for integrating pure functions. First, the notation.

```
In[52]:= Notation[ ∫& f_ ⇔ FunctionIntegral[f_] ]
```

We can now codify how FunctionIntegral works on pure functions. (Note the inter–mixing of notations.)

```
In[53]:= HoldPattern[∫& (λ_Symbol ↦ f_)&] :=
           (λ ↦ Evaluate@Simplify[∫ (λ ↦ f)& [λ] dλ])&

In[54]:= f

Out[54]= (x ↦ x^3 + Sin[2 x])&

In[55]:= ∫& f

Out[55]= (x ↦ 1/4 (x^4 - 2 Cos[2 x]))&
```

The previous examples concerning the creation of notations for integral-like functions are not as contrived as they might at first seem. Certain integrals are sometimes not easily solvable, and it is necessary to use a more specialized function, only applicable to certain classes of integrands. For two selected instances, see Feagin[107] AIV.2 and AIV.3, or Trott[312] problem 12.8a.

All the above notations were straightforward to declare using the function Notation. The notations that will be introduced in the rest of §2.2 *Notation* and in §2.3 *Symbolizations, Infix Notations, and InputAliases,* will also be easily declared. However, one must not be misled into thinking that any notation one can type into a notebook can so easily be made acceptable. Examples in this chapter, of notations which require more advanced techniques, will be given in §2.6.5 *Changing Grouping Behavior* and §2.7 *Bra-Ket Notation*. A reasonable treatment of tensors — such as that appearing in the next chapter, §3.4 *Tensorial Notation* — requires extensive use of such advanced techniques.

2.2.5 Notations that only Parse or only Format

So far, we have restricted consideration to Notation statements involving just '⟺', that is, notations which both parse input and format output. However, one can restrict a notation to only parse or only format by using '⟹' or '⟸', respectively, instead of '⟺', in the Notation statement.

First, we will describe notations that only parse, that is, notations of the form Notation[... ⟹ ...]. To illustrate this style of notation, let us add a new variant of our pure function notation introduced in the previous subsection.

```
In[56]:= Notation[x_ ↦ body_ ⟹ Function[x_, body_]]

In[57]:= f = {x, y} ↦ (z ↦ x^3 y^2 + z)

Out[57]= ({x, y} ↦ (z ↦ x^3 y^2 + z)&)&

In[58]:= f[a, b][c]
```

```
Out[58]= a^3 b^2 + c
```

Since $(x \mapsto f)_{\&}$ has already been defined as the notation for `Function[x, f]` in the previous section, it is appropriate that we only enter a parsing notation for $x \mapsto f$. (As an aside, for output formatting, we used the notation $(x \mapsto f)_{\&}$ in preference to $x \mapsto f$ since the former circumvents a bug in the parenthesization routines of *Mathematica* 3.0 and 4.0.)

The standard notation in *Mathematica* for partial differentiation is to use ∂, as in $\partial_x f$. With the use of a `Notation` statement, we can easily define another input notation for partial differentiation that looks more traditional, at least for n^{th} order partials with respect to the same variable.

```
In[59]:= Notation[ ∂f_/∂x_ ⟹ D[f_, x_] ]

In[60]:= Notation[ ∂^n_ f_/∂x_^n_ ⟹ D[f_, {x_, n_}] ]
```

If we wish, we can override how *Mathematica* normally formats the output of internal expressions such as `Derivative[n][f][x]` by the following.

```
In[61]:= Notation[ ∂f_[x_]/∂x_ ⟸ Derivative[1][f_][x_] ]
         Notation[ ∂^n_ f_[x_]/∂x_^n_ ⟸ Derivative[n_][f_][x_] ]
```

Derivatives are now formatted according to the new derivative notation.

```
In[63]:= {Γα'[t], (z + r)''[q]}

Out[63]= {∂Γα[t]/∂t, ∂^2 (r + z[q])/∂q^2}
```

To create a completely general definition to handle mixed partial derivatives using traditional notation requires more advanced techniques, covered in the next chapter (primarily the techniques to be presented in §3.3 *Complex Patterns in Notations*).

We have seen some examples of notation statements that only parse and some that only format. In §3.4 *Tensorial Notation*, we will see an important example where, instead of giving a single notation statement that both parses and formats, it is preferable to give separate statements for how to parse tensor input and how to format tensor output.

It is strongly recommended that notations should be defined, where possible, in such a way that they are usable for both input and output, since users will generally expect this functionality.

We close this section by reiterating our previous warning. For a syntactically complex notation, a natural tendency might be to try to save time and effort, even for creating one's first usage instance, by just copying the new notation from the defining notation statement and then pasting the copied pattern where one wants it, possibly as a template. However, this is very dangerous and should be avoided, for reasons which will become clear later. Be advised that, in general, if one tries a copy/paste operation based on the contents of a `Notation`, `Symbolize`, or `InfixNotation` statement, then a hidden and unwanted tag box may well be copied. Instead, if possible, use an input alias which we touched on in §2.2.2 *Notation: Examples*, and we treat in detail in §2.3.5 *Adding Input Aliases*.

2.3 Symbolizations, Infix Notations, and InputAliases

2.3.1 Symbolize

Let us start by first discussing exactly what constitutes a symbol in *Mathematica*. A symbol is a sequence of characters, each character being either a letter, a letter-like form, or a digit, with the first character being non–numeric. However, unlike many older languages, *Mathematica* is much more generous in which characters are allowed to make up a symbol. These characters include Greek, script, gothic, double struck, and extended Latin letters, as well as a large range of letter-like forms, such as various technical symbols, shapes, icons, and textual markers. (For a complete list of all letters and letter-like forms, see Wolfram[342] §*A.12 Listing of Named Characters*.) As an example of a bizarre symbol, consider

```
In[1]:= α2†𝕜5𝒦_wow // Head
Out[1]= Symbol
```

Users of mathematical notation often want to treat a 2-dimensional pattern of marks as an inseparable whole, as a name, a label. For instance, when considering 1-dimensional motion along a straight line, we might wish to represent the initial position and velocity of a particle by x_0 and v_0, respectively. For 3-dimensional motion, we might instead use $\overrightarrow{x_0}$ and $\overrightarrow{v_0}$, respectively. In quantum mechanics, a physicist might wish to denote the initial and final states of a system by ψ_i and ψ_f. In chemistry, one uses $\overline{H}_{\mathrm{id}}$ to denote the conventional molar enthalpy of an ideal gas. In finance, one uses r_f to denote the risk-free rate. In mathematics, we might wish to denote the zero of a special algebraic structure by 0*.

Mathematica usually parses 2-dimensional structures into 1-dimensional full form expressions which are not symbols. So, if we want to be able to treat a particular 2-dimensional structure as a symbol, that is, if we wish to symbolize a structure, then we must be able to over-ride *Mathematica*'s normal parsing behavior, and tell it instead to parse such a structure into a standard 1-dimensional symbol. The function `Symbolize` does exactly this.

`Symbolize[`*compositeBoxes*`]`	treat *compositeBoxes* as a symbol
`Symbolize[`*compositeBoxes*`, `*opts*`]`	treat *compositeBoxes* as a symbol with the options *opts*
`RemoveSymbolize[`*compositeBoxes*`]`	remove the treatment of *compositeBoxes* as a symbol

Syntax of symbolization declarations.

To illustrate the behavior of `Symbolize`, let us compare x_0 and y_0 after we have made use of the `Symbolize` template to symbolize x_0 but not y_0.

```
In[2]:= Symbolize[x0]
```

Now x_0 is treated as a symbol but y_0 is not, as we see by

```
In[3]:= Head /@ {x0, y0}
Out[3]= {Symbol, Subscript}
```

It is instructive to examine the symbol to which x_0 is parsed and the expression to which y_0 is parsed.

```
In[4]:= FullForm /@ {x0, y0}
Out[4]= {x_Subscript_0, Subscript[y, 0]}
```

Values can be assigned to both x_0 and y_0.

```
In[5]:= x0 := 4; y0 := 5
In[6]:= {x0 + 6, y0 + 6}
Out[6]= {10, 11}
```

The definition of x_0 can be queried in the normal way, since x_0 now parses to a symbol.

```
In[7]:= ? x0
        "Global`x_Subscript_0"
        x0 := 4
```

However, `? y0` is inadmissible since y_0 is a composite object and not a symbol.

```
In[8]:= ? y0
        Information::ssym : y0 is not a symbol or a string.
Out[8]= Information[y0, LongForm → False]
```

Regrettably, the quickest way to obtain the names and values of subscripted but non-symbolized expressions so far introduced is to use `?Subscript`.

Technical Note: Unfortunately, *Mathematica* is somewhat inconsistent in its handling of subscripts. Paralleling other *Mathematica* value functions, the designers ideally should have introduced a separate value function, say `ScriptValues`, to contain these values.

Before proceeding, let us clear the values of x_0 and y_0.

```
In[9]:= Clear[x0]; y0 =.
```

We cannot use `Clear[y0]` since y_0 is not a symbol. Of course x_0 still parses to a symbol. If desired, the symbolization for x_0 can be removed via `RemoveSymbolize`.

```
In[10]:= RemoveSymbolize[x0]
```

2.3.2 Symbolizing a Class of Expressions

`Symbolize` can also be used to symbolize an entire class of expressions. For example, sometimes we may want to collectively specify that all labels having the same name part but different subscripts should be treated as symbols. For instance, when considering a force field `F` in dynamics, one often uses F_x, F_y, F_z to denote the Cartesian components, and F_r, F_θ, F_ϕ to denote the spherical components, etc. We can collectively handle all these cases with the following single command, which symbolizes $F_{anything}$.

```
In[11]:= Symbolize[F_anything_]
```

We can now check whether various instances of $F_{any\_}$ are symbols, by checking if they have head `Symbol`.

```
In[12]:= Head /@ {Fx, F1+2}
Out[12]= {Symbol, Symbol}
```

Every new instance of a composite symbol gets a new full form name, as we see by the following.

```
In[13]:= {Fx, F1+2, Fx} // FullForm
Out[13]//FullForm=
        List[F_Subscript_x, F_Subscript_1_Plus_2, F_Subscript_x]
```

Since the 'x' appearing in a composite symbol such as 'F_x' is truly part of the symbol, we are also free to use 'x' again in other roles. For example, consider the following, which first defines and then calculates the force in the x-direction of an inverse-squared central force field.

```
In[14]:= Fx[x_, y_, z_] := k x / (x^2 + y^2 + z^2)^(3/2)

In[15]:= {Fx[2, 1, 2], Fx[x, 1, 2] /. x → 2}
```

Out[15]= $\left\{\frac{2\,k}{27}, \frac{2\,k}{27}\right\}$

Technical Note: A generalized symbolization statement such as `Symbolize[F_any_]` does not actually create any symbols; it just sets up rules for creating symbols, so that any expression matching `F_` is symbolized when entered. As currently implemented, the *Notation* package does not keep track of any symbols added by usage of a generalized symbolization, so that removing such a symbolization does not remove the symbols created by its usage.

Having considered symbolizations of the form `Symbolize[label_anything]`, let us next consider symbolizations of the form `Symbolize[anything_label]`. For example, it is common in science and engineering to denote the initial value of some quantity using a subscript '0'. We might wish to say that the initial coordinates of a particle are x_0, y_0, z_0, the initial electric field is $\vec{E}_0(\vec{r})$, etc.

```
In[16]:= Symbolize[any_0]
```

Every instance of a symbolized "initial value" symbol gets a new full form name, as we see by the following.

```
In[17]:= I_0 // FullForm
         E⃗_0 // FullForm
Out[17]//FullForm=
         I__Subscript__0
Out[18]//FullForm=
         E__Overscript__RightVector__Subscript__0
```

Creating a symbolization such as `Symbolize[_0]` introduces the potential for confusion. For instance, we can now have two expressions that have the same display form, but quite different full forms, as we next demonstrate.

```
In[19]:= {z_0, z_i /. i → 0}
Out[19]= {z_0, z_0}

In[20]:= FullForm /@ {z_0, z_i /. i → 0}
Out[20]= {z__Subscript__0, Subscript[z, 0]}
```

Although we have only given two forms of symbolizations in this subsection, it is important to point out that a symbolization can be as general as the patterns used inside the symbolization statement. We now turn our attention to one of the main reasons why it is sometimes necessary to have the capabilities of the `Symbolize` command.

2.3.3 Symbolized Structures as Pattern Variables

Though subscripted variables can not be used as pattern names in function definitions in *Mathematica,* symbolized expressions which look like subscripted variables can. Thus, `Symbolize` allows a user to create formulas in a more familiar notation. However, one must be careful to formulate pattern expressions in the form *CompositePatternVariable* : *PatternContent,* otherwise *Mathematica* will interpret the pattern expression as *CompositePatternVariable* * *PatternContent.* To clarify, let us symbolize $k_{any\_}$ and then examine patterns involving a subscripted k label.

```
In[21]:= Symbolize[k_any_]
```

```
In[22]:= k_1 _ // FullForm
Out[22]//FullForm=
Times[k_Subscript_1, Blank[]]
```

It is evident that k_1 _ is being interpreted not as a pattern labeled by k_1 but as k_1 times *anything.* This parsing problem can be avoided by using *symb* :_, which is equivalent to *symb*_ . We can confirm that k_1 :_ is interpreted as a pattern labeled by k_1 from the following.

```
In[23]:= k_1 :_ // FullForm
Out[23]//FullForm=
Pattern[k_Subscript_1, Blank[]]
```

Often in science, one defines certain functions in terms of subscripted variables. The corresponding function definitions in *Mathematica* would then be in terms of patterned subscripted variables. However, *Mathematica* does not allow unsymbolized objects to appear as names of patterns in the definition of a function, as is next illustrated.

```
In[24]:= f[x_1 :_] := Sin[x_1]
Syntax::sntxf : "f[" cannot be followed by "x_1 :_]".
f[x_1 :_] := Sin[x_1]
```

Using subscripted names which have been symbolized avoids this problem. For example, consider the following definition of the transmission coefficient for an incident wave packet on a potential step under certain physical conditions, where our subscripted names have already been symbolized.

```
In[24]:= transmissionCoefficient[k_1 :_, k_2 :_] := 1 - 4 k_1 k_2 / (k_1 + k_2)^2
```

We can see that this function works as intended.

```
In[25]:= transmissionCoefficient[α, β]
```

Out[25]= $1 - \frac{4\,\alpha\,\beta}{(\alpha+\beta)^2}$

Technical Note: In some versions of *Mathematica*, the option `ShortBoxForm` must be set to `False` in order to use pattern variables which are actually composite structures that have been symbolized. Otherwise, notebooks can become corrupted. This option can be set by using the option inspector in the **Format** menu. The option should be set to `False` at a notebook or global level.

For a more complex example using `Symbolize`, see §2.6.5 *Changing Grouping Behavior*.

Since many of the examples so far using `Symbolize` have involved subscripted variables, it must be pointed out that non-symbolized subscripted variables are equally important. For example, if one wants to manipulate a series of elements, then non-symbolized subscripted variables are necessary, as the following demonstrates.

```
In[26]:= Symbolize[z_]
```

In[27]:= $\left\{\sum_{j=0}^{3} y_j,\ \sum_{j=0}^{3} z_j\right\}$

Out[27]= $\{y_0 + y_1 + y_2 + y_3,\ 4\,z_j\}$

In summary, if one wants to refer to a subscripted variable x_2 as a special case of x_i, then one should not symbolize x_2. If one thinks of x_2 as a label or name, much like x2, then it may pay to symbolize it.

Before we continue, the general symbolizations of k and z, subscripted by anything, should be removed.

```
In[28]:= RemoveSymbolize[k_];
         RemoveSymbolize[z_];
```

2.3.4 Infix Notations

`InfixNotation` is the third fundamental function of the *Notation* package. It enables us to declare that a composite object, such as $+_{\mathcal{R}}$, will be treated as an infix notation. Although we can achieve this to a limited extent with a `Notation` statement, it will become apparent in §2.6.3 *InfixNotation Revisited*, that notation statements are not always sufficient.

`InfixNotation[` *infixOp*`,` *prefixHead*`]`	treat *infixOp* as an infix operator representing the function *prefixHead*
`InfixNotation[` *infixOp*`,` *prefixHead*`,` *opts*`]`	treat *infixOp* as an infix operator representing the function *prefixHead* with the options *opts*
`RemoveInfixNotation[` *infixOp*`,` *prefixHead*`]`	remove the composite infix operator *infixOp*

Syntax of infix notation declarations.

`InfixNotation` takes as arguments both a composite object and a symbol. The composite object represents the infix operator, and the symbol represents the full form *head* of the corresponding function. A simple parallel of this duality in *Mathematica* is that the infix notation '+' has the full form head `Plus`.

To illustrate `InfixNotation`, let us declare that the composite object $+_{\{\}}$ should act as an infix form of `Join`.

```
In[30]:= InfixNotation[+{}, Join]
```

Now, input involving $+_{\{\}}$ will be parsed to a kernel expression involving `Join`, and reciprocally a kernel expression involving `Join` will be formatted using $+_{\{\}}$. Moreover, this parsing and formatting will be done in keeping with the properties of `Join`, as is next illustrated.

```
In[31]:= {a, b} +{} {b, c} +{} {c, d}
Out[31]= {a, b, b, c, c, d}

In[32]:= Join[{a, b}, {b, c}, {c, d}] // HoldForm
Out[32]= {a, b} +{} {b, c} +{} {c, d}
```

Let us present another simple use of infix notation. Normally, *Mathematica* formats the function `Or` as `||`, as we see when we solve the following inequality.

```
In[33]:= << Algebra`InequalitySolve`

In[34]:= InequalitySolve[x (-3 + x^2) (-2 + x^2) > 0, x]
Out[34]= -√3 < x < -√2 || 0 < x < √2 || x > √3
```

Let us change the formatting of `Or` from `||` to the more mathematically standard symbol ∨.

```
In[35]:= InfixNotation[∨, Or]
```

Now the previous output is formatted with ∨.

```
In[36]:= %%
Out[36]= -√3 < x < -√2 ∨ 0 < x < √2 ∨ x > √3
```

Common infix notations which we will subsequently adopt are ≡ in place of `===` and ≢ in place of `!===`. Strangely, the input alias for '≡' is [ESC]===[ESC] and that of '≢' is [ESC]!===[ESC], yet there is no default interpretation for either ≡ or ≢.We fix this as follows.

```
In[37]:= InfixNotation[≡, SameQ]
         InfixNotation[≢, UnsameQ]
```

In certain circumstances, `InfixNotation` has major advantages over `Notation`. For instance, one might naively assume that it is possible to obtain the same change to the formatting of `Or` as above by using a notation statement like `Notation[x_ ∨ y_ ⟺ Or[x_, y_]]`. This and other circumstances necessitate the specialized function `InfixNotation`. These circumstances will be fully explained later, in §2.6.3 *InfixNotation Revisited.*

Technical Note: `InfixNotation` can be interpreted as a considerably more advanced formulation of the standard infix form of functions ~*f*~ — see the *Mathematica* Book 2.1.3 [342].

2.3.5 Adding Input Aliases

In the forgoing, we have only briefly touched on one of the important issues that faces the user when using the *Notation* package. Once notations have been defined, it is critically important that the user can enter them easily. The notations we will use can become exceedingly complex, as we will see in later sections. As has previously been mentioned in the earlier §2.2.2 *Notation: Examples*, we can add input aliases, so that a notation may be entered using an escape sequence, such as ESC*string*ESC.

`AddInputAlias[` *notation* `,` *string* `]`	add the notation *notation* to the input aliases so it can be entered by ESC*string*ESC

Syntax for adding input aliases.

As an example, we can easily create an input alias for the function `myApply`, created in §2.2.3 *Notation: Assignments*.

```
In[39]:= AddInputAlias[□ ⟶(Apply) □, "myApply"]
```

Then, entering ESC`myApply`ESC in a normal input cell would produce the following template.

$$\square \xrightarrow{\text{Apply}} \square$$

Next, we enter an input alias for the notation for pure functions created in §2.2.4 *Notation: Explicit Bracketing*.

```
In[40]:= AddInputAlias[(□ ↦ □)&, "function&"]
```

Finally, we can create an input alias for something that has nothing at all to do with any notations introduced by notation statements. Here is an input alias for an integral over momentum space.

```
In[41]:= AddInputAlias[∫ d^4 □ / (2 π)^4, "intd4"]
```

Unfortunately, input aliases are only added to the individual notebook in which the command `AddInputAlias` is executed, rather than the current front end session. It is not possible in *Mathematica* version 4.0 to add things to the front end session, although it is hoped that this feature may be available in later versions of *Mathematica*.

2.4 Options

The basics of the *Notation* package have now been covered. This section describes the options common to the functions `Notation`, `Symbolize`, and `InfixNotation`. These functions all take the options `WorkingForm` and `Action`. In addition, the *Notation* package has a local option, `AutoLoadNotationPalette`, which affects the loading of the palette.

2.4.1 The Option WorkingForm

option name	*possible values*	*effect*
`WorkingForm`	`StandardForm`	notations, symbolizations, and infix notations are defined in `StandardForm`
	`TraditionalForm`	notations, symbolizations, and infix notations are defined in `TraditionalForm`
	`Automatic`	notations, symbolizations, and infix notations are defined in the `DefaultOutputFormatType`

The working form option and some of its possible values.

The option `WorkingForm` specifies the *form* or *environment* in which a notation *works*, be it a `Notation`, `Symbolize` or `InfixNotation` statement. It does this by generating and adding rules to the grammar, rules which only work for input and/or output of the form specified by `WorkingForm`.

The typical values of `WorkingForm` are `StandardForm` or `TraditionalForm`. However, `WorkingForm` can be set to any form we define, for instance `LogicForm`, `PhysicsForm`, etc. One can view the current list of working forms by examining the value of `$BoxForms`. The mechanisms whereby a new working form can be defined are illustrated shortly.

When `WorkingForm` is set to `Automatic`, the working form will be the value of `DefaultOutputFormatType`. To see or set this front end option, one can use the menu item **Default Output Format Type** under the **Cell** menu. (For the purposes of this article, the default format types for both input and output are set to `StandardForm`.) The default value of `WorkingForm` is `Automatic`.

Let us illustrate these concepts by creating a `TraditionalForm` notation for the vector calculus functions `Div`, `Curl`, and the wrapper `Vector`. (Though not always necessary, the following `Notation` statements themselves are entered in traditional form.)

```
In[1]:= Notation[∇.expr_ ⟺ Div[expr_], WorkingForm → TraditionalForm]
        Notation[∇×expr__ ⟺ Curl[expr__], WorkingForm → TraditionalForm]
        Notation[v⃗_ ⟺ Vector[v_], WorkingForm → TraditionalForm]
```

These notations behave as expected: they only apply to expressions both entered and returned in `TraditionalForm`. For example, let us first enter $\nabla \times \vec{a}$ in `TraditionalForm`, and return the result in `TraditionalForm`.

```
In[4]:= ∇×a⃗ // TraditionalForm
Out[4]//TraditionalForm=
        ∇×a⃗
```

If we enter expressions in `TraditionalForm` but return the output in the default output format type, `StandardForm`, our expressions will be parsed but not formatted.

```
In[5]:= ∇×a⃗
Out[5]= Curl[Vector[a]]
```

As we see, the parsing rules for `Curl` and `Vector` were applied, but the formatting rules were not. Let us now enter a more complicated `TraditionalForm` expression and return, by default, a `StandardForm` result.

```
In[6]:= ∇.(∇×a⃗×b⃗) + ∇.(a⃗ + b⃗)
Out[6]= Div[Curl[Vector[a]×Vector[b]]] + Div[Vector[a] + Vector[b]]
```

However, as previously indicated, our notations only parse expressions entered in `TraditionalForm`. We can see this by trying to enter $\nabla \times \vec{a}$ in `StandardForm`.

```
In[7]:= ∇ × a⃗

        Syntax::sntxf : "∇" cannot be followed by "×a⃗".

        ∇×a⃗
```

If we want to create a notation that parses input entered in, say, $form_a$ and formats output in, say, $form_b$, then we can use a `Notation[ ... ⟹ ... ]` type statement with the option `WorkingForm` set to $form_a$ and a `Notation[ ... ⟸ ... ]` type statement with the option `WorkingForm` set to $form_b$.

To illustrate `WorkingForm` functioning with a non-standard form and also illustrate how to add a new form to the current list of working forms, let us declare that `MyEngineeringForm` has the parent form `StandardForm` and add it to the list of box forms.

```
In[7]:= wasProtected = Unprotect[ParentForm];
        ParentForm[MyEngineeringForm] = StandardForm;
        Protect[Evaluate[wasProtected]];
        AppendTo[$BoxForms, MyEngineeringForm];
```

Now we can enter new notations for `MyEngineeringForm`, just as we would for the forms `StandardForm` and `TraditionalForm`.

```
In[11]:= Notation[j ⟺ I, WorkingForm → MyEngineeringForm]

In[12]:= I + 1 // MyEngineeringForm

Out[12]//MyEngineeringForm=
         1 + j
```

The function `ParentForm` and the variable `$BoxForms` are minimally documented in *Mathematica,* and therefore only advanced users should try to use the above kind of functionality. However, for the advanced user, it is even possible to add this new form to the menus in the front end.

2.4.2 The Option Action

option name	*possible values*	*effect*
`Action`	`CreateNotationRules`	notation rules are entered into the system
	`RemoveNotationRules`	notation rules are removed from the system
	`PrintNotationRules`	cells containing the rules defining the given notation are printed into the current notebook

The action option and its possible values.

The `Notation`, `Symbolize`, and `InfixNotation` option `Action` determines what a notation statement does with the rules it generates. The default value of the `Action` option is `CreateNotationRules`, which causes the generated rules to be entered into the system.

The following statement creates a notation for iterated unions in keeping with sums and products.

```
In[13]:= Notation[ ⋃_{i_=m_}^{n_} expr_ ⟺ IteratedUnion[expr_, {i_, m_, n_}],
           Action → CreateNotationRules]
```

Iterated unions now use this new notation.

```
In[14]:= ⋃_{i=0}^{n} 𝒜_i // FullForm

Out[14]//FullForm=
         IteratedUnion[Subscript[𝒜, i], List[i, 0, n]]
```

If the `Action` option is set to `RemoveNotationRules`, then the notation rules are removed from the system. Using the option value `RemoveNotationRules` in a notation statement is equivalent to using the appropriate `RemoveNotation`, `RemoveSymbolize` or `RemoveInfixNotation` statement.

The following removes the above notation from the system.

```
In[15]:= Notation[ ⋃_{i_=m_}^{n_} expr_ ⟺ IteratedUnion[expr_, {i_, m_, n_}],
           Action → RemoveNotationRules]
```

Now the special notation for iterated unions is no longer defined.

```
In[16]:= ⋃_{i=0}^{n} 𝒜_i

Syntax::sntxi : Incomplete expression; more input is needed.

⋃_0^n 𝒜_i
```

By setting the option `Action` to `PrintNotationRules`, we can actually print out and view the rules generated by a notation statement. In fact, using this option in §3.2.1 *The Functioning behind the Notation Package*, we will be able to see how notation statements actually accomplish their intended behavior.

2.4.3 The Package Option AutoLoadNotationPalette

option name	*possible values*	*effect*
`AutoLoadNotationPalette`	`True`, *undefined*	the notation palette will be loaded when the *Notation* package loads
	`False`	the notation palette will not be loaded when the *Notation* package loads

The option to control automatic loading of the notation palette.

If one is designing a new package that relies on the *Notation* package, it may be desirable that the notation palette does not load when the new package loads (so as not to confuse users). This is accomplished by setting `AutoLoadNotationPalette` to `False` inside the new package as follows:

```
Utilities`Notation`AutoLoadNotationPalette = False
```

If the value of `AutoLoadNotationPalette` is undefined or set to `True`, then the notation palette will be loaded when the new package loads the *Notation* package.

2.4.4 The Symbolize Option SymbolizeRootName

Unless directed otherwise, `Symbolize` will automatically generate the full form name to be used for a symbolized version of a particular structure. Sometimes the user may wish to directly specify the full form name. This is the purpose of the option `SymbolizeRootName`.

option name	*possible values*	*effect*
`SymbolizeRootName`	`""` (the default)	`Symbolize` will generate a new symbol name
	string	`Symbolize` will use *string* as the root of the new symbol name

The option for generating a new root name when symbolizing and its behavior.

For instance, in electromagnetism two constants arise: the *permittivity* ϵ_0 and *permeability* μ_0 *of free space*. Let us first symbolize ϵ_0 and μ_0 without making use of the above option.

```
In[17]:= Symbolize[ϵ0]
         Symbolize[μ0]

In[19]:= FullForm /@ {ϵ0, μ0}

Out[19]= {ϵ$Subscript$0, μ$Subscript$0}
```

It is evident that the symbol names that the *Notation* package chose were not close to any descriptive names a physicist might use. Using the option `SymbolizeRootName` we can choose more appropriate names as follows.

```
In[20]:= Symbolize[ϵ0 , SymbolizeRootName → "PermittivityFS"]
         Symbolize[μ0 , SymbolizeRootName → "PermeabilityFS"]
```

Now the full forms of these symbols are more recognizable.

```
In[22]:= 1/√(ϵ0 μ0) // FullForm

Out[22]//FullForm=
         Power[Times[PermeabilityFS, PermittivityFS], Rational[-1, 2]]
```

2.5 Box Structures

2.5.1 Introduction to Box Structures

Now that the basic ideas and functions of the *Notation* package have been presented, we should briefly describe how *Mathematica* actually represents expressions in the front end. At a fundamental level, all *Mathematica* front end input and output is made up of box structures, that is, nested collections of boxes. When one enters a parsable textual expression into *Mathematica*, the corresponding box structure is parsed into an internal full form expression, internal evaluation then occurs, and finally the resulting internal expression is transformed back into a box structure for displaying in the *Mathematica* front end. For example, consider the expression

$$\frac{a_i}{3^2 + 1} \tag{2.5.a}$$

Internally, the front end stores this expression as

```
FractionBox[SubscriptBox["a", "i"],
   RowBox[{SuperscriptBox["3", "2"], "+", "1"}]]
```
(2.5.b)

This can be revealed by using the Show Expression command from the Format menu of the front end. If the text of an input cell constitutes an *admissible* (or *acceptable*) expression, that is, one that can be parsed to a full form, then another way of seeing its box structure is to apply the function `MakeBoxes` to it.

```
In[1]:= MakeBoxes[ a_i / 3^2 + 1 ]

Out[1]= FractionBox[SubscriptBox["a", "i"],
          RowBox[{SuperscriptBox["3", "2"], "+", "1"}]]
```

`MakeBoxes` takes an acceptable kernel expression and essentially converts it to its box structure *without evaluation*. Reciprocally, to see how the front end will display a given box structure we can use `DisplayForm`.

```
In[2]:= % // DisplayForm

Out[2]//DisplayForm=
        a_i
      ------
      3^2 + 1
```

`ToBoxes`, a function similar to `MakeBoxes`, also shows the box structure of an acceptable kernel expression, but *after evaluation*.

```
In[3]:= ToBoxes[ a_i / 3^2 + 1 ]
```

```
Out[3]= FractionBox[SubscriptBox["a", "i"], "10"]
```

At this stage, it is suggested that the reader should become familiar with the contents of §2.8.3 *The Representation of Textual Forms* and §2.8.10 *Representing Textual Forms by Boxes*, of the *Mathematica* Book [342]. These sections give several examples describing the boxes listed above. However, let us give a brief encapsulation of the information contained in these sections by looking at some simple box structures using `MakeBoxes`.

`SubscriptBox` is used to represent a subscript.

```
In[4]:= MakeBoxes[α_i]
Out[4]= SubscriptBox["α", "i"]
```

Note that admissible symbols such as Greek letters are treated in a box structure in the same way as symbols consisting of ordinary alphanumeric characters.

`FractionBox` is used to represent a fraction.

```
In[5]:= MakeBoxes[𝒜/ℬ]
Out[5]= FractionBox["𝒜", "ℬ"]
```

`RowBox` is used to group together a sequence of boxes and/or strings appearing in an expression.

```
In[6]:= MakeBoxes[α^2 + β_]
Out[6]= RowBox[{SuperscriptBox["α", "2"], "+", "β_"}]
```

From this, it is evident that the expression $\alpha^2 + \beta\_$ is represented as a `RowBox` containing a list of three items: a `SuperscriptBox` structure followed by two strings.

Most of the other boxes have similarly intuitive meanings. For example, a `FrameBox` puts a visible frame around another box structure, while a `StyleBox` modifies the style of the box structure it surrounds, say by turning the text green and using a different font.

In *Mathematica* versions 3.0 and 4.0, the accepted fundamental boxes are `SubscriptBox`, `SuperscriptBox`, `SubsuperscriptBox`, `UnderscriptBox`, `OverscriptBox`, `UnderoverscriptBox`, `FractionBox`, `SqrtBox`, `RadicalBox`, `RowBox`, `GridBox`, `FrameBox`, `InterpretationBox`, `FormBox`, `StyleBox`, `AdjustmentBox`, `ErrorBox`, `ButtonBox`, and `TagBox`.

`TagBox` figures prominently in the rest of this chapter, and indeed thesis, and lies at the heart of the *Notation* package. The *Mathematica* Book [342] contains no information of substance on tag boxes, which is extremely unfortunate since they are vital to achieving any significant notational changes, as we will see in later sections.

Let us comment on what takes place in the *Mathematica* front end during editing. As one edits an expression on-screen, the internal box structure is being correspondingly manipulated. Consider the following incomplete input.

$$\frac{a_i}{3^{2+}}$$

Internally, the front end stores this incomplete expression as a box structure; and as one edits this textual expression, say typing '1' after the '+', the internal box structure is correspondingly manipulated. Any *Mathematica* textual expression in an input cell, *even an ill-formed or incomplete one*, has a corresponding *box structure*, that is, a nested collection of boxes.

```
In[7]:= Times[x, y] // FullForm
Out[7]//FullForm=
        Times[x, y]
```

Consider the box structures of the textually similar expressions x^2, x^+ and $x^\pm$. The first is both admissible, i.e. parsable, and meaningful. The second is acceptable but not meaningful — for example, 3^+ has no defined value. The third is not even normally admissible. Yet, all three have corresponding box structures. In fact, they have the same basic box structure, namely `SuperScript["x", script]`, where *script* is '2', '+', or '±'. To reiterate, one must not confuse a textual expression being admissible with its having a box structure.

Having minimally discussed box structures, we can elucidate the functioning of `Symbolize`. Let us symbolize $\mathcal{L}^{-1}$. This does not change the box structure of $\mathcal{L}^{-1}$ into that of a true symbol, as we see from the following.

```
In[8]:= Symbolize[ℒ^-1, SymbolizeRootName -> "LInverse"]
        MakeBoxes[ℒ^-1]
Out[9]= SuperscriptBox["ℒ", RowBox[{"-", "1"}]]
```

However, it does change what the box structure of $\mathcal{L}^{-1}$ parses to, and that indeed is a symbol; and conversely that symbol formats to $\mathcal{L}^{-1}$.

```
In[10]:= ℒ^-1 // FullForm
Out[10]//FullForm=
         LInverse
```

2.5.2 Parsing and Formatting

Let us take a look at the overall process of going from an input expression to a corresponding output result. As has been pointed out, each structured expression is stored in the front end in terms of a box structure. When the *Mathematica* kernel receives input from the front end, the input box structure is transformed into a kernel expression. Evaluation then takes place within the kernel. Finally, the resulting internal expression is transformed back into a box structure for display in the *Mathematica* front end if an output is returned. Transforming a box structure to a kernel expression is referred to as *parsing*, and the reverse, transforming a kernel expression to a box structure, is referred to as *formatting*.

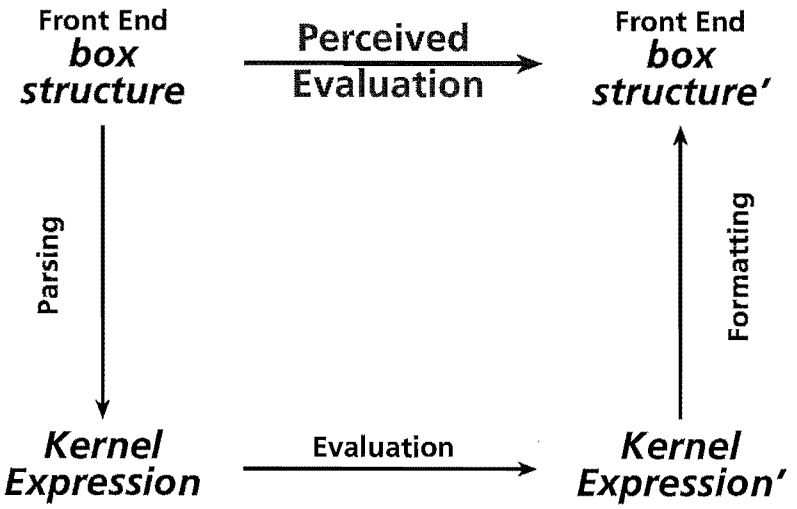


Figure 2.5.A : The process of parsing, evaluation, and formatting.

The function that carries out parsing is `MakeExpression`. It is called when the front end passes a box structure to the kernel for eventual evaluation. `MakeExpression` creates a kernel full form expression from the supplied box structure. For example

```
In[11]:= MakeExpression[RowBox[{"a", "+", RowBox[{"3", " ", "4"}]}]] //
          FullForm
Out[11]//FullForm=
         HoldComplete[Plus[a, Times[3, 4]]]
```

The `HoldComplete` in the output is present to ensure that no evaluation of the expression takes place during parsing. The parsing process is quite separate from the evaluation process. (The `FullForm` is included so we can see the structure of the result.)

The function that carries out the formatting is `MakeBoxes`, which we have already mentioned in the previous section. `MakeBoxes` is normally called once an evaluation is complete and the resulting expression needs to be transformed into a box structure to be sent back to the front end for display. For example

```
In[12]:= MakeBoxes[Plus[a, Times[3, 4]]]
Out[12]= RowBox[{"a", "+", RowBox[{"3", " ", "4"}]}]
```

`MakeBoxes` and `MakeExpression` are inverses of each other. We can represent their behavior by the following diagram.

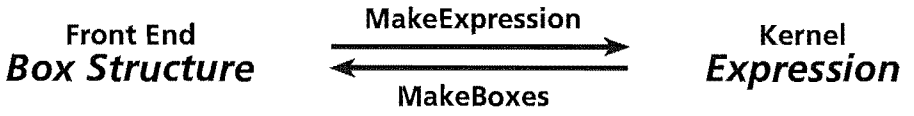


Figure 2.5.B : Reciprocal nature of `MakeExpression` and `MakeBoxes`.

One important factor in the above model is that the *Mathematica* front end normally strips superfluous spacing characters, styling information, and other boxes that do not change the underlying expression before the box structure is even passed to `MakeExpression`. Therefore, it is generally not possible to distinguish notations on the basis of italics or other stylistic features that are stripped out. For example, it would not be easy, if at all possible, to set up a notation to recognize bold objects as vectors. (However, one could use tag boxes to visually emulate this. Better still, one could use tag styles — see §2.6.7 *Style Sheets and Tag Styles*.)

Finally, it is clear from the above diagram that `MakeBoxes` applies just to kernel expressions. Yet, in the previous subsection, we had many examples of what appeared to be the application of `MakeBoxes` to "2-D expressions" which *appeared* not to be kernel expressions. But there is no conflict. To see this, consider `MakeBoxes` $[\frac{a_i}{3^2+1}]$. The box structure of this gets passed to `MakeExpression`, which results in

```
In[13]:= HoldComplete[
           MakeBoxes[Times[Subscript[a, i], Power[Plus[Power[3, 2], 1], -1]]]]
```

Out[13]= HoldComplete[MakeBoxes[$\frac{a_i}{3^2+1}$]]

Evaluation then strips the `HoldComplete` and `MakeBoxes` then returns its result. Thus, `MakeBoxes` really is operating on kernel expressions, even though it may have appeared otherwise in the previous subsection.

2.5.3 Modifying the Grammar

What makes a given input inadmissible is that for some part of the input, there is no corresponding `MakeExpression` statement which matches it, hence no rule telling the kernel how to convert this part into an admissible kernel expression. For example, standard *Mathematica* parses structures matching x_⊕y_ but not x_ +$_{\mathcal{G}}$ y_ because *Mathematica* includes a `MakeExpression` rule for the former but not the later.

By creating new rules for `MakeExpression` and `MakeBoxes`, new parsing and formatting behaviors can be added. This is how the functions `Notation`, `Symbolize`, and `InfixNotation` of the *Notation* package are able to introduce new notations: essentially they generate and add some additional rules for `MakeExpression` and/or `MakeBoxes`, effectively extending the grammar of *Mathematica*.

The concepts introduced in the previous subsection are illustrated in the *Mathematica* Book, §2.8.17 *Advanced Topic: Low-Level Input and Output Rules*, by stepping through an example of adding a notation by hand. We will now quickly present a similar example, adding the operator notation $+_{\mathcal{R}}$ to the grammar. Let us first confirm that the current grammar of *Mathematica* does not accept `a` $+_{\mathcal{G}}$ `b`.

```
In[14]:= a +ℊ b

        Syntax::sntxi : Incomplete expression; more input is needed.

        a ±ℊ b
```

Let us now add a rule to `MakeExpression` that allows *Mathematica* to parse this class of expressions.

```
In[14]:= MakeExpression[RowBox[{x_, SubscriptBox["+", "ℊ"], y_}],
              StandardForm] := MakeExpression[
              RowBox[{"GroupPlus", "[", RowBox[{x, ",", y}], "]"}],
              StandardForm]
```

Now *Mathematica* can parse the input `a` $+_{\mathcal{G}}$ `b`.

```
In[15]:= a +ℊ b

Out[15]= GroupPlus[a, b]
```

However, the situation is slightly different for the formatting of output. Any kernel output can always be formatted using the current rules of `MakeBoxes`. For example, if the result of a kernel evaluation is `GroupPlus[2, 3]` and if no special `MakeBoxes` rules are present, then the standard `MakeBoxes` rules will apply and the output will be `GroupPlus[2, 3]`. However, if one adds a new `MakeBoxes` rule which `GroupPlus[2, 3]` matches, then this new rule would apply. Let us now add such a new `MakeBoxes` rule, one which will give us the desired formatting of `GroupPlus` objects.

```
In[16]:= MakeBoxes[GroupPlus[x_, y_], StandardForm] :=
           RowBox[{MakeBoxes[x, StandardForm],
                 SubscriptBox["+", "ℊ"], MakeBoxes[y, StandardForm]}]
```

Now `GroupPlus` objects are formatted according to the desired notation.

```
In[17]:= GroupPlus[a, b]

Out[17]= a +ℊ b
```

To reiterate, `Notation`, `Symbolize`, and `InfixNotation` statements essentially define new parsing behavior by creating appropriate `MakeExpression` rules, and similarly define new formatting behavior by creating appropriate `MakeBoxes` rules.

2.5.4 Tag Boxes in Notation Statements

So far we have considered examples of box constructs, such as `RowBox`, `FractionBox`, `SubscriptBox`, and `SuperscriptBox`. Let us now consider tag box structures, that is, box structures with the head `TagBox`. A tag box structure is of the form `TagBox[`*boxes*`,` *tag*`,` *options*`]`. Tag boxes are used to indicate or change the grouping, encapsulation, or interpretation of a subexpression at an underlying level. Tag boxes involving *tag* can be intercepted before conventional parsing occurs, and instead parsed according to additional rules for `MakeExpression` that involve *tag*. Such an interception is essential for allowing the input and functioning of `Notation`, `Symbolize`, and `InfixNotation` statements.

Technical Note: Usually the tag in a tag box is a symbol, although this is not a requirement. The tag can actually be any expression, which in particular, includes strings. This is important for defining notations in the context of packages. This point will be revisited later in §3.2.3 *Tags in Tag Boxes*.

In this subsection, we will only be concerned with showing how tag boxes occur in a notation statement. To this end, let us examine the box structure of the template `Notation[□ ⟺ □]`. Using Show Expression, we see that, apart from tag box options (which differ between the versions of *Notation* for *Mathematica* 3.0 and 4.0), it is

```
RowBox[{"Notation", "[",
   RowBox[{
     TagBox["□", NotationBoxTag],
     " ", "⟺", " ",
     TagBox["□", NotationBoxTag]}], "]"}]
```
(2.5.c)

Observe that each placeholder '□' is embedded in a corresponding tag box structure with the tag `NotationBoxTag`. These tag boxes allow the contents of the relevant '□' to be intercepted and treated according to rules associated with `NotationTagBox` rather than as the parser would normally treat them. Examining the box structures of the other *Notation* palette templates would reveal that the exact same tag box structure surrounds the placeholder in `Symbolize[□]` and the first placeholders in `InfixNotation[□, □]` and `AddInputAlias[□, □]`.

Any expression matching *expr*$^{\pm}$ is normally unrecognizable by the *Mathematica* parser. To make it recognizable, let alone define it to be something meaningful, we will need to set up rules allowing such an expression to be parsed, and furthermore, dictate the form to which it is parsed. As can be gathered from what has been said so far, the *Notation* package will generate such rules when we supply it with a notation statement like

```
Notation[x_± ⟺ superPlusMinus[x_]]
```
(2.5.d)

But how can the *Notation* package compile such a statement, since it contains the so far unrecognizable subexpression `x_±`, hence it would seem that the whole notation statement would be unrecognizable until the rules for recognizing it were generated? This is where tag boxes are pivotal. Since the `Notation` statement above was entered using the palettes, `x_±` is embedded in a tag box structure. This allows `x_±` to be intercepted before normal parsing to a full form is attempted. The tag `NotationBoxTag` present in the tag box "directs" *Mathematica* how to "handle" `x_±`. What actually happens during and after interception will be discussed in §3.2 *Principles behind the Notation Package.*

Sometimes it is necessary to be able to modify an input cell by adding hidden tag boxes to it manually. To accomplish this, enter the input as normal and then, using Show Expression, convert the cell to show its box form. Next, edit the existing box structure, adding the desired tag boxes. Finally, using Show Expression again, convert the cell back to ordinary input.

2.6 Precedence and Grouping of Operators

In this section, we will first consider how *Mathematica* normally structures input, depending on the precedence and grouping of the objects that make up the input. We will then consider how one can alter this behavior when desired. In particular, we will make use of the encapsulation nature of tag boxes.

2.6.1 Precedence and Grouping of Characters and Operators

So far, we have not discussed the relationship between the visual screen form of textual input and its corresponding box structure. This relationship is determined entirely by the front end, even before one hits Enter or Shift-Return. All input cell expressions, no matter how outlandish, have a corresponding box structure, but obviously not all box structures are parsable to kernel expressions.

The precedence and grouping behavior of operators is very important at the front end level. To illustrate this, consider `a +ℛ b * c`.

```
In[1]:= a +ℛ b * c

        Syntax::sntxi : Incomplete expression; more input is needed.

        a ±ℛ b * c

        a ±ℛ b * c
```

Clearly, a +ℛ b * c is not currently parsable. However, using Show Expression, we can see that it does have a corresponding box structure, namely

```
RowBox[
   {"a", SubscriptBox["+", "R"], RowBox[{"b", "*", "c"}]}]
```
(2.6.a)

We see that b is grouped with c rather than with a, hence the grouping is like (a +ℛ (b * c)). This is because * has a higher precedence than +ℛ, for reasons to be explained; so * is first to pick up its operands, namely b and c, essentially giving what is equivalent to (b * c). Then +ℛ gets to pick up its operands, which are now a and (b * c).

In mathematics, the relative precedences of some operators, and consequently how strings of characters will be grouped, is sometimes left open. A typical instance of this would be $\nabla \times \vec{a} \times \vec{b}$. To specify groupings when there is no accepted convention, the user must include sufficient parentheses or other bracketing so as to allow parsing — for example, to write $(\nabla \times \vec{a}) \times \vec{b}$ or $\nabla \times (\vec{a} \times \vec{b})$, whichever one is meant. (Even if not needed, a little bracketing is not only safer, but also makes for more readable notation.)

In contrast, each character or operator in *Mathematica* has a relative precedence and a grouping behavior. For example, ∏ groups to the right and has a higher precedence than ⊕, which is an infix grouping operator that in turn has a higher precedence than many other operators, for instance +. As has already been pointed out, these grouping and relative precedence relations take effect during front end editing. To illustrate this, using Show Expression one could check that the expression ∏ a +ℛ f⊕b is front end internally grouped like ((∏ a) +ℛ (f⊕b)), though neither expression is parsable to a kernel expression.

Front end structuring incorporates the correct behavior of *flat* (*non-flat*)operators, resulting in flat (non-flat) structures. (Intuitively, flat objects are non-nested — see §2.5.3 *Attributes* in the *Mathematica* Book [342].) For example, using Show Expression, one can check that a+b+c and a^b^c are respectively structured as

```
RowBox[{"a", "+", "b", "+", "c"}]
```
(2.6.b)

```
RowBox[{"a", "^", RowBox[{"b", "^", "c"}]}]
```
(2.6.c)

Hence, the front end structuring of a+b+c is flat whereas the structuring of a^b^c is not. The grouping nature and relative precedence of all *Mathematica* operators and symbols is given in Appendix A.2.7 *Operator Input Forms,* in the *Mathematica* Book [342].

2.6.2 Precedence and Grouping of Composite Structures

When it comes to handling a composite operator, even if it does not constitute a parsable notation, the front end precedence/grouping rules are determined according to the components from which the operator is composed. For instance, the operator $\oplus_n^*$ has the same front end precedence / grouping as does $\oplus$, and the mapping $\xrightarrow{\texttt{myApply}}$ has the same front end precedence / grouping as does $\longrightarrow$. Generally speaking, the front end precedence / grouping of a composite operator is determined by the operator on which it is based. For instance, `SubscriptBox[`*base*`,` *subscript*`]` has the same front end precedence and grouping behavior as does *base*. Therefore, $*_{\mathcal{R}}$ has a higher precedence than + or $+_{\mathcal{R}}$ since * has a higher precedence than +. Finally, several notations with the same "base", such as + and $+_{\mathcal{R}}$, are treated as having equal front end precedence. This is illustrated by the fact that the box structure of a + b $+_{\mathcal{R}}$ c is

```
RowBox[{"a", "+", "b", SubscriptBox["+", "ℛ"], "c"}]
```
(2.6.d)

The design decision to make the front end precedence / grouping of compound operators depend upon their constituents makes intuitive sense and generally leads to notations that are consistent with one another. For instance, assume we wish to work with a ring $\langle \mathcal{R}, +_{\mathcal{R}}, *_{\mathcal{R}} \rangle$. Even before we make the composite notations $+_{\mathcal{R}}$ and $*_{\mathcal{R}}$ for ring addition and ring multiplication acceptable to *Mathematica*, a typical front end expression such as a $+_{\mathcal{R}}$ b $*_{\mathcal{R}}$ c will have a box structure essentially equivalent to that of (a $+_{\mathcal{R}}$ (b $*_{\mathcal{R}}$ c)). This grouping is due to, as one would expect, $*_{\mathcal{R}}$ having precedence over $+_{\mathcal{R}}$ since * has precedence over +. Let us now make these composite notations admissible.

```
In[1]:= Notation[x_ +ℛ y_ ⟺ RingPlus[x_, y_]]
        Notation[x_ *ℛ y_ ⟺ RingTimes[x_, y_]]
```

Mathematica can now parse expressions containing $+_{\mathcal{R}}$ and $*_{\mathcal{R}}$.

```
In[3]:= a +ℛ b *ℛ c // FullForm
Out[3]//FullForm=
        RingPlus[a, RingTimes[b, c]]
```

Also the reverse process, formatting expressions containing `RingPlus` and/or `RingTimes` in terms of the composite notations $+_{\mathcal{R}}$ and $*_{\mathcal{R}}$, works properly. The output has the correct formatting, styling, spacing, and parenthesization:

```
In[4]:= RingTimes[RingPlus[a, b], c]
Out[4]= (a +ℛ b) *ℛ c
```

2.6.3 InfixNotation Revisited

As seen in §2.6.1 *Precedence and Grouping of Characters and Operators*, the structuring of a+b+c is flat in the front end. And, as explained in §2.6.2 *Precedence and Grouping of Composite Structures*, the precedence and grouping of $+_{\mathcal{R}}$ is the same as that of +. Hence, we also expect a $+_{\mathcal{R}}$ b $+_{\mathcal{R}}$ c to have a flat structure in the front end. Indeed, by examining the box structure, we can confirm this.

```
RowBox[{"a", SubscriptBox["+", "ℛ"],
      "b", SubscriptBox["+", "ℛ"], "c"}]
```
(2.6.e)

Therefore, we would expect that an expression involving say just the operator $+_{\mathcal{R}}$ and a similar expression involving just + would parse to similar full forms. But they do not.

```
In[5]:= a + b + c // FullForm
        a +ℛ b +ℛ c // FullForm
Out[5]//FullForm=
        Plus[a, b, c]
Out[6]//FullForm=
        RingPlus[RingPlus[a, b], c]
```

Clearly the grouping of the notation for `RingPlus` that we defined in the preceding section is not working in the way that we desire. That a $+_{\mathcal{R}}$ b $+_{\mathcal{R}}$ c has a flat box structure in the front end is in stark contrast to the fact that it parses like (a $+_{\mathcal{R}}$ b) $+_{\mathcal{R}}$ c. This problem arises since the notation defined for `RingPlus` only works with two arguments, whereas the normal notation defined for `Plus` of course works with multiple arguments.

```
In[7]:= {RingPlus[a, b, c], Plus[a, b, c]}
Out[7]= {RingPlus[a, b, c], a + b + c}
```

In general, a `Notation` statement creates a notation which only works with the explicit number of arguments given in the `Notation` statement. This behavior of `Notation` is the main reason for the genesis of the more specialized function `InfixNotation`. An `InfixNotation` statement creates a notation which works correctly with any number of arguments. To demonstrate this, let us first remove the `Notation` statements for $+_{\mathcal{R}}$ and $*_{\mathcal{R}}$, and replace them with corresponding `InfixNotation` statements.

```
In[8]:= RemoveNotation[x_ +ℛ y_ ⟺ RingPlus[x_, y_]]
        RemoveNotation[x_ *ℛ y_ ⟺ RingTimes[x_, y_]]
        InfixNotation[+ℛ, RingPlus]
        InfixNotation[*ℛ, RingTimes]
```

Now let us give some examples illustrating that the new notations handle more than two arguments in the way desired.

```
In[12]:= {a +ℛ b +ℛ c // FullForm, RingPlus[a, b, c]}
```

```
Out[12]= {RingPlus[a, b, c], a +ℛ b +ℛ c}

In[13]:= a +ℛ b +ℛ c *ℛ d *ℛ e +ℛ f // FullForm
Out[13]//FullForm=
        RingPlus[a, b, RingTimes[c, d, e], f]
```

In summary, in our usage an *infix notation* is more than just a notation for a binary operator whose front end form appears in an infix position. It must also result in parsing and formatting of expressions in keeping with the parsing, formatting, grouping, and precedence properties one expects of the underlying n-ary operator.

2.6.4 How Boxes can Affect Grouping

The grouping behavior of some boxes can be affected by the grouping behavior of their contents. To illustrate this, we will consider the grouping behavior of `SubscriptBox`. Assume that in an input cell, one has so far entered the following.

```
a + b □ℛ c
```
(2.6.f)

This has the box structure

```
RowBox[{"a", "+",
        RowBox[{"b", SubscriptBox["□", "ℛ"], "c"}]}]
```
(2.6.g)

If we enter '*' in the placeholder '□' in the front end expression (2.6.f), we get something with box structure

```
RowBox[{"a", "+",
        RowBox[{"b", SubscriptBox["*", "ℛ"], "c"}]}]
```
(2.6.h)

If we instead enter '+' in the placeholder '□', we get something with the flat box structure

```
RowBox[{"a", "+", "b", SubscriptBox["+", "ℛ"], "c"}]
```
(2.6.i)

Thus, it is clear that the grouping behavior of `Subscript[`*base*`,` *subscript*`]` is indeed affected by the grouping behavior of *base*. But for some structural boxes, the grouping behavior is not thus affected. For instance, the grouping behavior of `FractionBox[`*numerator*`,` *denominator*`]` is independent of the grouping behavior of *numerator* and *denominator*.

The following table shows which boxes can affect the grouping behavior of surrounding elements.

type of box	*behavior*
`SubscriptBox`, `SuperscriptBox`, `SubsuperscriptBox`, `UnderscriptBox`, `OverscriptBox`, `UnderoverscriptBox`, `AdjustmentBox`, `ErrorBox`, `StyleBox`, `FrameBox`, `TagBox`	These boxes do not isolate their contents from the outside; the contents of these boxes can affect how surrounding elements are grouped
`FractionBox`, `RadicalBox`, `SqrtBox`, `RowBox`, `GridBox`, `ButtonBox`, `FormBox`, `InterpretationBox`	These boxes isolate their contents from the outside; the contents of these boxes do not affect how surrounding elements are grouped

The standard boxes and their relationship to grouping.

Knowledge of the above grouping nature of boxes is important when it comes to changing the grouping behavior of an operator, as we will see in the next subsection.

2.6.5 Changing Grouping Behavior

As previously pointed out, when one enters text in an input cell, the front end structures this internally as a box structure. Loading the *Notation* package and declaring notations will in *no way* affect how the contents of an input cell will be structured in terms of boxes. In any given box structure, all structuring is explicit. To reiterate, this structuring is due to the precedence and grouping of the constituent parts of the box structure. So just how, if at all, can we change the precedence and/or grouping behavior of an operator? In §2.7 *Example: Bra-Ket Notation*, which implements the bra-ket notation of quantum mechanics, we will provide a very practical example of such a desirable change. In this subsection, we will consider two simpler examples in order to give some idea of how to make such changes.

Say we would like to introduce yet another notation for pure functions, one similar to *Mathematica*'s standard version which uses '&', but more suggestive. Let us attempt to use $f\,\&_x$ in place of `Function[x, f]`. For, example, we might use $x^2 + 3\&_x$ instead of `#^2 + 3 &` to denote the function given by the rule $x \mapsto x^2 + 3$. Normally $f\,\&_x$ will group as $(f\ \&)_x$. In fact, if we even try to type $f\,\&_x$ into the front end, it will structure itself as $(f\ \&)_x$. We can, however, rectify this by wrapping a tag box around the $\&_x$, since a tag box encapsulates its contents. So when wrapped, it would have the following underlying structure.

```
TagBox[SubscriptBox["&", "x"], "functionTag"]                    (2.6.j)
```

The tag name `"functionTag"` was chosen to be suggestive of its purpose, though we could have chosen any symbol or string not already involved in parsing and/or formatting. Using the box structure (2.6.j), we next create our desired notation for pure functions.

```
In[14]:= Notation[body_&λ_ ⟺ Function[{λ}, body_]]
```

Now pure functions are formatted and parsed in this new notation.

```
In[15]:= Function[{x, y}, x^2 + y^2]
```

```
Out[15]= x^2 + y^2 &x,y
```

Contrast this with $\#1^2 + \#2^2$ &, the way this pure function would usually have been previously entered in standard *Mathematica*.

```
In[16]:= x^2 + y^2 &x,y // FullForm
Out[16]//FullForm=
        Function[List[x, y], Plus[Power[x, 2], Power[y, 2]]]
```

The new notation is easily able to represent functions of functions.

```
In[17]:= x^2 y &x &y // FullForm
Out[17]//FullForm=
        Function[List[y], Function[List[x], Times[Power[x, 2], y]]]
```

There is, of course, the question of how we can conveniently enter the above inputs involving $\&_{var}$, since they contain hidden tag boxes. We will discuss this difficulty in §2.7.3 *Implementing the Notation*. For now, however, we offer an appropriate input alias, usable in *Mathematica* 4.0.

```
In[18]:= AddInputAlias[□&□, "function"]
```

It should be noted that in constructing our new function notation above, we used the tag box structure (2.6.j), which has the string tag `"functionTag"`. Up to this point, the only tags in the tag box structures have all been symbols. In contrast, when creating tag box structures to be used in notations, it is sometimes desirable to use tags which are strings. The advantage of, and reasons for, this will be fully explained in §3.2.3 *Tags in Tag Boxes*. Briefly though, the main reason to choose a string in preference to a symbol is that when a notation is defined in one context, it should work equally well in other contexts; and this is sometimes not so if a symbol is used. Henceforth, in this chapter and indeed throughout the rest of this thesis, in tag box structures used for notations, we will usually use tags which are strings.

We next consider another, somewhat more complex, example. Say we would like to introduce a symbol with ∂ appearing not as an operator but as a character at the end. To be specific, say we want to join the symbol `generic` and the operator character ∂ to form the symbol `generic`∂. Unfortunately, just typing in the eight characters appearing in `generic`∂ will result in an unparsable input expression that is structured in the front end like `RowBox[{ "generic", "`∂`" }]`, hence not structured as a single composite object.

Moreover, if we tried `Symbolize[generic`∂`]`, it would fail because structures to be symbolized cannot have a `RowBox` as their head. `Symbolize` must be used only on a single composite object, for example something whose head is `SuperscriptBox`, `SubscriptBox`, `OverscriptBox`, etc. Importantly, `Symbolize` can also be used on a box structure whose head is `TagBox`, since the effect of a `TagBox` is to encapsulate the expression it surrounds into a single object.

A further complication is that ∂ groups to the right. Say we want to use our new generic–partial and enter something like `generic`∂ `f`. Unfortunately this is structured in the front end as

```
RowBox[{"generic", RowBox[{"∂", " ", "f"}]}]
```
(2.6.k)

since the operator ∂ naturally groups to the right. Thus we need to suppress the grouping behavior of ∂. This can be accomplished by encapsulating `RowBox[ { "generic", "∂" } ]` inside a `TagBox`. In addition, if we include a small space between `generic` and ∂, then we can also suppress the visual space. Finally, we can add suitable tag box options to ensure that the new object, once created, will not be editable nor will subparts of it be selectable. Therefore, a suitable solution is

```
genericD                                                    (2.6.1)
```

whose box structure has been edited manually to conform to the following.

```
TagBox[RowBox[{"generic", "\[VeryThinSpace]", "\[PartialD]"}],
   "genericPDTag", Editable -> False, Selectable -> False]
```

It only remains to declare that the encapsulated object (2.6.1) be treated as a symbol.

```
In[19]:= Symbolize[genericD]
```

If we wish to, we can now attach rules to the composite symbol `genericD`. Recall that in §2.2.2 *Notation: Examples*, we defined a notation for generic integrals that looked like $\int_{\mathcal{R}} f$. Even though this should just be considered a toy example, we could nevertheless add a toy fundamental theorem for domain integrals.

```
In[20]:= genericD /: ∫_{D[R_]} genericD[f_, R_] := f
```

Here is an example making use of the above rule.

```
In[21]:= ∫_{D[R+]} genericD[f[x, y], R+]

Out[21]= f[x, y]
```

To recap, in this section we have used the encapsulating nature of a `TagBox` to alter normal grouping behavior.

2.6.6 Changing Precedences and the Option SyntaxForm

In *Mathematica* 3.0.1, `SyntaxForm` was added to the list of tag box options. Using this option, one can change the precedence of an operator contained in a tag box. A tag box containing a `SyntaxForm` option will look like `TagBox[` *box structure*`,` *tag*`, SyntaxForm →` *string*`]`, where *string* is a string indicating the operator on which the precedence of the tag box is to be modelled.

To illustrate the use of the `SyntaxForm` option, first consider the following example which shows that the standard arrow "↔" has a precedence higher than we sometimes want.

```
In[22]:= a + b ↔ c // FullForm
Out[22]//FullForm=
        Plus[a, LongLeftRightArrow[b, c]]
```

Thus, `a + b ↔ c` is being grouped as `a + (b ↔ c)` whereas we would like it to group as `(a + b) ↔ c`. Using a Notation statement, we can define a new composite arrow which is still displayed as '↔' but which has a much lower precedence, say that of ',', hence groups as desired. This is accomplished by surrounding the '↔' in the defining notation statement by a tag box that has the `SyntaxForm` option set to ','. Specifically, the substructure `TagBox[ "↔", Identity, SyntaxForm → "," ]` is embedded in the following notation.

```
In[23]:= Notation[a_ ↔ b_ ⟹ LongLeftRightArrow[a_, b_]]
```

This new composite arrow now groups as desired.

```
In[24]:= (a + b ↔ c) // FullForm
Out[24]//FullForm=
        LongLeftRightArrow[Plus[a, b], c]
```

We illustrate the underlying groupings of the expressions above in the following table.

visual form	*grouping*	*box form*
`a+b ↔ c`	`a+(b↔c)`	`RowBox[{"a","+",RowBox[{"b","↔","c"}]}]`
`a+b ↔ c`	`(a+b)↔c`	`RowBox[{RowBox[{"a","+","b"}], TagBox["↔",Identity,SyntaxForm->","],"c"}]`

A table illustrating the precedences and grouping of expressions with and without precedence changing tag boxes.

The `SyntaxForm` option value must be a string. This string can consist of any single operator character present in the `UnicodeCharacters.tr` file — see [316]. It can also include symbols before and after the operator to indicate whether the precedence is that of a prefix operator, an infix operator, or a postfix operator. Some typical values for the `SyntaxForm` option are given in the table below.

`SyntaxForm` *value*	*precedence behavior*
`"*"`	group as the operator times
`"a"`	group as a symbol
`"a+b"`	group as an infix plus operator
`"∀"`	group as a for all operator
`"∫"`	group as an integrate operator
`"⋃ a"`	group as a prefix union operator
`" "`	group as white space

Typical syntax form values and their associated precedence behaviors.

For a "real world" example, consider the interior product, '⌋', in differential geometry — see Crampin & Pirani [78]. The *interior product* of v and ω is the $(p-1)$-form $v \rfloor \omega$ (read "v hook ω") defined by

$$(v \rfloor \omega)(v_1, v_2, \ldots, v_{p-1}) = \omega(v, v_1, v_2, \ldots, v_{p-1}) \tag{2.6.m}$$

However, as soon as we try to enter any expression with a '⌋' in *Mathematica*, even ignoring the unmatched bracket style, the grouping is very different from what we want. For instance, the following input

```
ω1 + b⌋ ω
```

is grouped like $(\omega_1 + b) \rfloor \omega$. This grouping is clearly wrong. To rectify the grouping, we can wrap the '⌋' character with a tag box having the option `SyntaxForm → "*"`. This ensures that the overall structure has an infix grouping with the precedence of multiplication. Thus, for our underlying box structure, we choose the following.

```
TagBox[StyleBox["⌋",
      AutoStyleOptions → {"UnmatchedBracketStyle" → None}],
   "InteriorProductTag", SyntaxForm → "*",
   Editable → False, Selectable → False]
```
(2.6.n)

Using (2.6.n) as the underlying box structure for our "interior product" operation, we create the following notation.

```
In[25]:= Notation[v_ ⌋ ω_ ⟺ InteriorProduct[v_, ω_]]
```

Here is a simple example demonstrating the correct parsing, grouping, and precedence of the interior product operation.

```
In[26]:= a + InteriorProduct[v, ω + μ]
Out[26]= a + v ⌋ (μ + ω)
```

Here is a simple rule involving the linearity of the interior product.

```
In[27]:= v_ ⌋ (ω_ + μ_) := v ⌋ ω + v ⌋ μ

In[28]:= a + InteriorProduct[v, ω + μ]
Out[28]= a + v ⌋ μ + v ⌋ ω

In[29]:= FullForm @ %
Out[29]//FullForm=
Plus[a, InteriorProduct[v, μ], InteriorProduct[v, ω]]
```

Operations like this are used in coordinate free calculations in differential geometry. For more information on this, see the REDUCE program EXCALC[286, 287, 288].

2.6.7 Style Sheets and Tag Styles

From the material in the previous subsections, it should have become apparent that it is sometimes necessary to specify several different styles/options for some particular box. This need will become apparent when we introduce Dirac's "bra-ket notation" in the next section. We will need to specify several options for our tag box wrappers and also for our argument wrappers. We could do this in one of two ways: include all of the options in each particular box structure, or modify the box structure options according to a *named style*. Up until now, we have only described how to use the first approach. Below we touch upon the second approach, the one we will mostly use in the future.

Adding a new *named style* to the style sheet allows us to set up a uniform style that will be used consistently throughout a notebook. (The reader can access the style sheet in *Mathematica* through the Edit Style Sheet... command in the Format menu of the front end.) A named style can be modified at any time, resulting in the changes to the style being immediately applied uniformly throughout the document. It should thus be obvious that it is preferable, when possible, to define named styles rather than specifying isolated style options in a haphazard way as one encounters the need for them. Indeed, we will find that it is necessary to use named styles in some of the sections to come, such as §2.7 *Example: Bra-Ket Notation* and §3.4 *Tensorial Notation*. For a thorough discussion of style sheets and other front end issues, consult Gray [132].

For now though, let us demonstrate how named styles are used. Assume, for instance, that we want to be able to have some letters boldfaced in input and output (maybe to represent a vector or otherwise). We proceed by adding the following named style to the style sheet. Actually, the following style has been previously added to the style sheet, but it looks like the following.

Prototype for style: "BoldTagStyle":
BoldTagStyle

If one were to examine its underlying structure using the Show Expression command from the Format menu, it would look like the following.

```
Cell[StyleData["BoldTagStyle"],
  StyleMenuListing->None,
  FontWeight->"Bold"]
```

Once a style, *style name*, has been added to the style sheet, any box structure that is surrounded by a `StyleBox[`*box expr*`, "`*style name*`"]` will inherit all the options defined in *style name*. Moreover, in *Mathematica* 4.0 we can use the `TagStyle` option of a tag box. For instance, any box structure `TagBox[`*box expr*`,` *tag*`, TagStyle →` *"style name"*`]` will inherit all the options defined in *style name*.

`BoldTagStyle` sets the `FontWeight` option to `Bold`. In addition, the `BoldTagStyle` also sets the option `StyleMenuListing` to `None`, indicating that this style should *not* appear on the list of styles in the front end. However, we could have set a whole host of options for this style. In fact, we could have changed almost any of the styles present in the **Option Inspector**. Among the options that are extremely important to us are the following: `Editable`, `Selectable`, and `SyntaxForm`.

To make use of the bold style, all we have to do is either encase the structure we want with a `StyleBox[ … , "BoldTagStyle" ]` or a `TagBox[ … , "boldTag", TagStyle → "BoldTagStyle"]`. Setting the tag style option is the preferable approach. By setting this, we do not have to worry about style boxes being accidently deleted or propagated or otherwise. Furthermore, since we normally have to use tag boxes anyway in our structures, we can simplify the underlying structure by using a tag box with a tag style option rather than surrounding a tag box with a style box.

For instance, say we want to symbolize all bold variables. We can easily achieve this by the following `Symbolize` statement.

```
In[30]:= Symbolize[expr_]
```

where the underlying structure of the boxes being symbolized is

```
TagBox["expr_", "boldTag", TagStyle -> "BoldTagStyle"]
```
(2.6.o)

When we use "bolded" characters in some input (that is, characters wrapped in a `boldTag` tag box), they remain bolded in the output.

```
In[31]:= 2 (4 w + 3 v) // Expand

Out[31]= 6 v + 8 w

In[32]:= 6 v + 8 w // FullForm

Out[32]//FullForm=
        Plus[Times[6, v$1], Times[8, w]]
```

Moreover, the underlying structure is relatively simple. As always, we can of course add an appropriate input alias.

```
In[33]:= AddInputAlias[□, "bold"]
```

After the above statement has been executed, we can enter ESC bold ESC in any notebook that has the bold style in its style sheet, and we will obtain a placeholder surrounded by the appropriate bold box structure.

If we symbolize a structure of the form (2.6.o), then we obtain something which has head `Symbol` (we could also consider this to have the "type" `Symbol`). Alternatively, for various reasons, we might wish certain bold characters to have the "type" (or head) `Vector`, or maybe a more specific vector type, such as `Momentum`, `Force`, `ElectricField`. For example, to give a notation for vectors we could enter a notation statement like `Notation[`*`expr_`*` ⟺ Vector[`*`expr_`*`]]`, where the left hand side is wrapped with a `boldTag` tag box. In this case, however, we probably would have named our style `VectorStyle` and our tag, `vectorTag` or something similar.

In fact, the above paragraph underscores an important point. It is strongly recommended that one use different styles for different objects, since at some later stage one may need to change one style independently of the other. For instance, in the above example, which used the bolded style for vectors, it may be necessary at a later date to distinguish between vectors and some other unforeseen character which also needs to be bolded. Hence, one might try and use a different styling for vectors, say italicization (for example, I.B.1 of [69]); but since just a general bold style was used instead of a specific `VectorStyle`, it is not possible to modify the vectors in isolation.

As users of *Mathematica* 4.0 have no doubt observed, the placeholders appearing in the templates for `Notation`, `Symbolize`, `InfixNotation`, and `AddInputAliases` are shaded yellow. Such shading is simply achieved by appropriately setting the tag style option for the tag boxes embedded in the templates — see §2.5.4 *Tag Boxes in Notation Statements*. We will enter into the functioning of the package in §3.2 *Principles behind the Notation Package*.

Currently, it is not easy to programmatically add styles to the style sheet. Hopefully, this deficiency will be rectified in a future release of *Mathematica*. The author has long advocated the use of *cascading style sheets* to the designers of *Mathematica*. Cascading style sheets are basically multiple style sheets with a set precedence hierarchy for determining which style sheet to use if a style appears in more than one style sheet. Not surprisingly, it is acknowledged amongst the front end design staff of Wolfram Research Institute that such a feature is highly desirable. Sadly, such a neat solution is not yet available. Nevertheless, the resolute and determined package designer can use a private function, `AddStylesToNotebook`, which is defined inside the *Notation* package, to programmatically add styles to the style sheet. However, the use of this private function is not for the uninitiated and should only be attempted by the expert, and somewhat masochistic, *Mathematica* practitioner.

Now that tag styles and style sheets have been introduced, and the reader has an inkling of how they are used in order to set up uniform and consistent styles, we can proceed on to the example of bra-ket notations. This example will make use of a large amount of the information given so far in a concrete and practical setting.

2.7 Example: Bra-Ket Notation

Now that we know how to use the *Notation* package and have some understanding of how the package works at a low level, we can present an example illustrating all the forgoing sections. In this section, we will develop a non-trivial *Notation* example that uses tag boxes to modify grouping and editing behaviors. Of course, once a notation has been created, others can copy and use the notation, including all its hidden structure and associated styles (contained in the style sheet), if any, without knowing the details of the notation definitions or how it was constructed.

Dirac's bra-ket notation for the state vectors of a quantum mechanical system is an indispensable notation for physicists. Consequently, a visually and structurally correct implementation of the bra-ket notation has long been coveted by physicists using *Mathematica* or one of the other computer algebra systems. Such an implementation is presented in this section, using just the knowledge and techniques of the previous sections. Actually, a non-physicist does not need to know any quantum mechanics or even any physics to follow a large part of this section — namely §2.7.1 *Prototypical Ket Structures*, §2.7.2 *Prototypical Ket Structures using Named Styles*, and §2.7.3 *Implementing the Notation* — since the focus in these subsections will be on notation. However, for a reference, see any book on quantum mechanics — for example, Cohen-Tannoudji [68] Chapter II Section B).

2.7.1 Prototypical Ket Structures

We first consider what is called the *ket* state vector $|\psi\rangle$ for a quantum mechanical state ψ. It should be obvious that *Mathematica* will not even parse the notation '$|\psi\rangle$' without special intervention, since it includes a right angle bracket not balanced by a left angle bracket.

```
In[1]:= | ψ⟩

        Syntax::bktmop : Expression " | ψ⟩" has no opening "⟨".

        | ψ⟩
```

There is yet another difficulty. Say we want to represent a multiplication of a constant, say $\hbar$, times the ket vector $|\psi\rangle$. One would enter this as

```
ħ | ψ⟩
```

But this has the box structure

```
RowBox[{RowBox[{"ħ", " ", "|", "ψ"}], "⟩"}]
```

It is clear that '$|\psi$' has been grouped with '$\hbar$' rather than '$\rangle$' since the precedence of '$|$' is higher than that of '$\rangle$'. This is not what we want. We could resort to putting brackets around all our kets, as in $\hbar(|\psi\rangle)$, but this would make our notation *highly* non–standard; moreover, it would have other parsing problems as well.

We want the symbols "$|$", "ψ" and "$\rangle$" to be grouped together as a single structure, a ket structure. To accomplish this desired grouping and precedence changing, we will use tag boxes. As a first attempt, we might use a box structure like the following.

```
In[1]:= TagBox[ RowBox[{"|", "ψ", "⟩"}], "Ket"];
```

This box structure is displayed as desired.

```
In[2]:= % // DisplayForm
Out[2]//DisplayForm=
        | ψ⟩
```

The fact that $|\psi\rangle$ is actually a composite object, consisting of a tag box structure, is not immediately visible to the user. This box structure has some of the desired properties: it is grouped into a single encapsulated object and its display form looks correct on screen (except for the bracket matching). To use this box structure in a functioning notation, all that would remain would be to enter a statement like `Notation[|ψ__⟩ ⟺ Ket[ψ__]]`. In practice, there are some further details that we must be concerned with, and we now address these.

We must ensure that the user can only edit the argument of the ket and not delete the '|' or the '⟩'. Therefore, we must add the option `Editable → False` to the outer tag box to make the structure uneditable. However, we must be able to edit the argument of the ket; so in addition, we must introduce an inner tag box, surrounding just the argument of the ket and having the option `Editable → True`. So motivated, our next attempt might be

```
TagBox[
   RowBox[
      {"|", TagBox["ψ", "KetArgs", Editable -> True], "⟩"}],
   "Ket", Editable -> False]
```
(2.7.a)

Also, we must not allow selection of only part of the ket, or else the user might not copy the vital tag box information, and hence the new input would not be able to be interpreted as a ket. Therefore, we must add the option `Selectable → False` to the outer tag box. However, as before, we must allow the argument of the ket to be selectable, so we add the option `Selectable → True` to the inner tag box. Thus, our next candidate is

```
TagBox[
   RowBox[{ "|", TagBox["ψ", "KetArgs",
            Editable -> True, Selectable -> True], "⟩"}],
   "Ket", Editable -> False, Selectable -> False]
```
(2.7.b)

However, it is also desirable to add some styling changes to ensure that the '|' and the '⟩' are slightly larger than normal, and that auto highlighting of unmatched brackets is turned off. So, the next candidate would have the following structure.

```
TagBox[
   RowBox[{
         StyleBox["|", SpanMinSize -> 2, AutoStyleOptions ->
               {"UnmatchedBracketStyle" -> None}],
         TagBox["ψ", "KetArgs", Editable -> True,
            Selectable -> True],
         StyleBox["⟩", SpanMinSize -> 2, AutoStyleOptions ->
               {"UnmatchedBracketStyle" -> None}]}],
   "Ket", Editable -> False, Selectable -> False]
```

We must also ensure the overall structure acts like a symbol, so we must set the syntax form of the outer tag box to a single symbol, as described in §2.6.6 *Changing Precedences and the Option SyntaxForm*. The final problem, which is only observable under close inspection, is that the ket argument is slightly too close to the '⟩'. We rectify this by adding an adjustment box that moves the argument over by a small amount. (We could have added thin spaces to move the argument but, for reasons that will become clear, it is better to use an adjustment box.) Together, these changes yield the following candidate.

```
TagBox[
   RowBox[{
         StyleBox["|", SpanMinSize -> 2, AutoStyleOptions ->
               {"UnmatchedBracketStyle" -> None}],
         AdjustmentBox[ TagBox["ψ", "KetArgs",
               Editable -> True, Selectable -> True],
            BoxMargins -> {{-0.1, 0.1}, {0, 0}}],
         StyleBox["⟩", SpanMinSize -> 2, AutoStyleOptions ->
               {"UnmatchedBracketStyle" -> None}]}],
   "Ket", Editable -> False, Selectable -> False,
   SyntaxForm -> "symbol"]
```

(2.7.d)

Undeniably, the box structure candidate above has become fairly complex. The box structure can be made far simpler by placing the various styles in a style sheet, thus uniformly incorporating all of the above information. The next subsection does exactly this.

2.7.2 Prototypical Ket Structures using Named Styles

In the previous subsection, we progressively built up the tag box structure for a prototypical ket vector (2.7.d). As was apparent, the prototypical structure was fairly complex. The solution to this predicament is to use named styles to replace all of the style options set in the tag boxes. This can be done in the way previously discussed in §2.6.7 *Style Sheets and Tag Styles*. By suitably defining the styles `BraKetArg` and `KetWrapper`, we can simplify the prototypical box structure (2.7.d) to obtain our new distilled prototypical ket structure (2.7.e).

```
TagBox[
   RowBox[{"|", AdjustmentBox[
            TagBox["ψ", "KetArgs", TagStyle -> "BraKetArg"],
            BoxBaselineShift -> 0], "⟩"}], "Ket",
   TagStyle -> "KetWrapper", SyntaxForm -> "symbol"]
```
(2.7.e)

The approach of using named styles, however, requires the addition of several styles to the style sheet. The styles that we need to add are the following ones.

Prototype for style: "BraKetArg":
BraKetArg

Prototype for style: "KetWrapper":
KetWrapper

Were these styles to be examined, one would find that they have the following structures.

```
Cell[StyleData["BraKetArg"],
  AutoStyleOptions->{"UnmatchedBracketStyle"->"UnmatchedBracket"},
  SpanMinSize->Automatic,
  StyleMenuListing->None,
  AdjustmentBoxOptions->{BoxMargins->{{0, 0}, {0, 0}}},
  TagBoxOptions->{Editable->True,
  Selectable->True}]
```

```
Cell[StyleData["KetWrapper"],
  AutoStyleOptions->{"UnmatchedBracketStyle"->None},
  SpanMinSize->2,
  StyleMenuListing->None,
  AdjustmentBoxOptions->{BoxMargins->{{-0.1, 0.1}, {0, 0}}},
  TagBoxOptions->{Editable->False,
  Selectable->False,
  SyntaxForm->"symbol"}]
```

By inspecting these styles and our new prototypical ket structure (2.7.e), it should be apparent how we have avoided explicitly setting all of the options present in the earlier candidates, but have instead incorporated these options into the named styles. All the options that are set in the wrapping tag box are reset in the internal tag box that surrounds the argument. For instance, `Editable` is set to `False` in `KetWrapper` but it is reset to `True` inside `BraKetArg`; similarly for `Selectable`, `SpanMinSize`, `AutoStyle`, and the adjustment box `BoxMargins`. In this way the named styles embody all of the explicitly set options in the box structure (2.7.d).

Two last quandaries remain. (*i*) Why do we explicitly include the option `SyntaxForm → "symbol"` in our prototypical ket (2.7.e)? (*ii*) Why do we have the adjustment box in our prototypical ket (2.7.e) when it only appears to set the option `BoxBaselineShift`? The answer to the first question is that in *Mathematica* 4.0.1 there is currently a bug in the internal parenthesizing routine. If the syntax form option is set in the style sheet, it is not obeyed; however if it is set explicitly in the box structure, it is obeyed. Thus, our explicit inclusion of the syntax form option is necessary to circumvent this bug. The resolution of point (*ii*) is that *Mathematica* will unfortunately delete an adjustment box from a box structure if no options are given to the adjustment box. Thus, we need to include some dummy adjustment to stop the automatic removal of the adjustment box. A much nicer solution would be for the front end to include box margins as a basic option to be set at the normal style level, for example, something of the form `StyleBox[ …, BoxMargins → {{l, r}, {b, t}}, AdjustMargins → True ]`. In this way, we could include the adjustment right into our named styles, and our prototype would consequently become

```
TagBox[
   RowBox[{"|",
         TagBox["ψ", "KetArgs", TagStyle -> "BraKetArg"], "⟩"}],
   "Ket", TagStyle -> "KetWrapper"]
```
(2.7.f)

This would be truly elegant. Alas, it will have to wait for a future version of *Mathematica*.

The distilled prototypical ket structure (2.7.e) only works correctly with *Mathematica* 4.0 and later, since *Mathematica* 3 did not implement tag styles. *Mathematica* 3.0.0 did not even possess the ability to have one tag box inside another with the appropriate handling of the options. This situation was rectified in *Mathematica* 3.0.1. Thus, it was still possible in *Mathematica* 3.0.1 to use named styles to greatly simplify the structure of (2.7.d). Indeed, the following box structure (2.7.g) is the more refined ket structure that one would use when named styles are allowed but the `TagStyle` option is not allowed.

```
TagBox[
   StyleBox[
      RowBox[
         {"|", AdjustmentBox[StyleBox[TagBox["ψ", "KetArgs"],
                  "BraKetArg"], BoxBaselineShift → 0], "⟩"}],
      "KetWrapper"],
   "Ket", Editable → False,
   Selectable → False, SyntaxForm → "symbol"]
```
(2.7.g)

It would not be possible, providing *Mathematica* functions correctly, to delete or edit either of the style boxes in structure (2.7.g). One of the great advantages of using the `TagStyle` option is that one avoids any complications arising from making sure that the compensating style boxes cannot be deleted and do not extend themselves. For the remainder of this section, we will assume that the user is working with *Mathematica* 4.0, unless otherwise stated.

Previously, in §2.2.3 *Notation: Assignments*, we introduced the notation { *f* | *domain, condition* }. It should be evident that some of the difficulties encountered in developing a "proper" notation for kets are also present in our previous presentation of `ListOfAll`, especially those related to `Editable` and `Selectable`. Fortunately, it should be just as evident how the techniques illustrated in this subsection, especially the use of named styles, can be adapted to finding a safer implementation of `ListOfAll`.

After the protracted development of the prototypical ket structure, it now remains to actually enter the notation.

2.7.3 Implementing the Notation

We now use the underlying box structure of a ket, developed in the previous subsections, for the left hand side of a `Notation` statement for ket structures.

```
In[3]:= Notation[ |ψ__⟩ ⟺ Ket[ψ__]]
```

Using **Show Expression**, one can check that our ket notation statement contains our prototypical ket structure with the embedded tag boxes. Our notation now correctly implements the physicist's ket notation.

```
In[4]:= Ket[r, l, m]
Out[4]= |r, l, m⟩

In[5]:= ħ |α, r, l, m⟩ // FullForm
Out[5]//FullForm=
        Times[ħ, Ket[α, r, l, m]]
```

In a similar way, we can implement the notation for bras and bra-kets. The underlying box structures for bras and bra-kets exactly parallel that of kets.

```
In[6]:= Notation[⟨ψ__| ⟺ Bra[ψ__]]

In[7]:= Notation[⟨ ψf__ | ψi__ ⟩ ⟺ BraKet[{ψf__}, {ψi__}]]
```

Just as the notation for ket used the named style `KetWrapper`, our above notations use corresponding named styles, `BraWrapper` and `BraKetWrapper`. These styles have also been preloaded.

The full forms of expressions involving bras, kets or bra-kets are uniform and intuitive.

```
In[8]:= BraKet[{ψf}, {ψi}] ħ
Out[8]= ħ ⟨ ψf | ψi ⟩
```

One final issue remains. The bra, ket, and bra-ket structures are made up with complex underlying tag boxes. The structures for tensors, to be introduced in §3.4.2 *Prototypical Tensor Box Structure*, are even more complex. So how is the average user meant to enter expressions involving such notations in order that they may be used? In the case of *Mathematica* 4.0, there is fortunately a more natural solution than for *Mathematica* 3.0. As has been indicated in §2.3.5 *Adding Input Aliases*, we can use input aliases. Consequently, we add the following aliases to the current notebook.

```
In[9]:= AddInputAlias[ |□⟩, "ket"]
        AddInputAlias[⟨□ |, "bra"]
        AddInputAlias[⟨□ | □⟩, "braket"]
```

In the case of *Mathematica* 3.0, the task is awkward. To be specific, let us consider entering kets. One method is, every time we want to enter a ket, we enter it in full form, for example as `Ket[ψ]`; then select this expression, and then under the **Cell** menu, select **ConvertTo** → **StandardForm** (even if **StandardForm** already has a check mark). (If we work in traditional form, then of course we should select **TraditionalForm**.) Obviously, this method will only work if the ket notation is active within the current *Mathematica* session. A second approach would be to copy an existing ket from somewhere else, possibly from within our current notebook, and then edit this to the desired form. However, in *Mathematica* 3.0, attempting to do this by trying to copy a prototype from within a `Notation` statement is very dangerous and should be avoided, since the prototype will necessarily have a hidden `NotationBoxTag` surrounding it, which might be inadvertently copied.

We should lastly note that in the package which we create based on the above information, that is the *BraKet* package, we define the underlying structures slightly differently than those given in this chapter. The difference is not critical, and only constitutes the inclusion of an additional overall adjustment box in the underlying structure. This additional adjustment box allows us to control the space to the right and to the left of the bra / ket structures. However such an inclusion would add little to the above discussion since the concepts that are being explained should by now be clear.

2.7.4 Example Calculations from Physics

The readers that are familiar with physics will understand the following steps; those that are not conversant in physics should try to observe how the notations are seamlessly interoperating within the calculations. The toy rules that are set up in this section are really only for illustrative purposes. Once we have covered the underlying language modifications in §4 *Language Modifications*, we will provide a suitable semantics for our bras and kets. Indeed, the following section could be viewed as the way calculations were done before the upcoming language modifications were developed.

Let us also introduce some notations for operators and eigenkets. (Note how the second notation below is defined in terms of the first notation.)

```
In[12]:= Notation[Subscript[$\hat{\mathcal{H}}$, _] ⟺ Operator[Subscript[$\mathcal{H}$, _]]]
```

```
In[13]:= Notation[Subscript[$\lambda$, Subscript[$\hat{\mathcal{H}}$, _]] ⟺ EigenLabel[$\lambda$_, Subscript[$\hat{\mathcal{H}}$, _]]]
```

Technical Note: Although I will not enter into the issue of representations in detail at this stage, in some cases the notations a physicist uses can be ambiguous. A computer scientist might say that physicists do not lexically scope their notations. At best, they have semantic meaning beyond any syntactic representation. For instance, consider the kets |x⟩, |i⟩ and |p⟩. In each of these states, the corresponding letter represents some specific semantic variable that is not free to change. For example, we could not use the ket |x⟩ to represent an arbitrary momentum state vector, even though there is no syntactic reason preventing this.

Let us introduce some basic results from angular momentum theory. For reference, see any text on quantum mechanics — for instance, Shankar[292], Cohen-Tannoudji[68], or more specifically, Wiesbluth[334], Thompson[311], or Fano[106]. Firstly, we will denote a simultaneous eigenstate of the operators $\hat{J}^2$ and $\hat{J}_z$ by the angular momentum ket $\left|j_{\hat{J}},\ m_{\hat{J}_z}\right\rangle$. Secondly, let us denote the operation of `NonCommutativeTimes` by '$\cdot$', and give it the attributes of `Flat` and `OneIdentity`.

```
In[14]:= InfixNotation[·, NonCommutativeTimes];
         SetAttributes[NonCommutativeTimes, {Flat, OneIdentity}];
```

Technical Note: One applies `SetAttributes` to the full form name, not the new infix operator notation.

Thirdly, for the sake of style, let us symbolize the raising and lowering operators J_+ and J_- as well as the operator J_z.

```
In[15]:= Symbolize[J+]; Symbolize[J-]; Symbolize[Jz]
```

Finally, we can define how our raising and lowering operators act on eigenstates of angular momentum.

```
In[16]:= Subscript[$\hat{J}$, z] · |Subscript[j, $\hat{J}$], Subscript[m, Subscript[$\hat{J}$, z]]⟩ := m ħ |Subscript[j, $\hat{J}$], Subscript[m, Subscript[$\hat{J}$, z]]⟩
```

```
In[17]:= Subscript[$\hat{J}$, +] · |Subscript[j, $\hat{J}$], Subscript[m, Subscript[$\hat{J}$, z]]⟩ := ħ Sqrt[j (j + 1) - m (m + 1)] |Subscript[j, $\hat{J}$], (m + 1) Subscript[$\hat{J}$, z]⟩
```

```
In[18]:= Subscript[$\hat{J}$, -] · |Subscript[j, $\hat{J}$], Subscript[m, Subscript[$\hat{J}$, z]]⟩ := ħ Sqrt[j (j + 1) - m (m - 1)] |Subscript[j, $\hat{J}$], (m - 1) Subscript[$\hat{J}$, z]⟩
```

We can now perform some simple calculations.

```
In[19]:= Subscript[$\hat{J}$, +] · |Subscript[3, $\hat{J}$], Subscript[0, Subscript[$\hat{J}$, z]]⟩
```

```
Out[19]= 2 Sqrt[3] ħ |Subscript[3, $\hat{J}$], Subscript[1, Subscript[$\hat{J}$, z]]⟩
```

Again, without entering into detail about how to represent operators at a lower level or how to properly handle operator computations in *Mathematica*, we must add some basic properties to our `NonCommutativeTimes`. The following is an extremely crude implementation of multiple linearity over addition and over constants; however, it is sufficient for the calculations in this section. The following rules state that constants can be factored out of a `NonCommutativeTimes` expression, and that `NonCommutativeTimes` distributes over addition.

```
In[20]:= ℓ___ · (c_?ConstQ a_) · r___ := c ℓ · a · r
         ℓ___ · (a_ + b_) · r___ := ℓ · a · r + ℓ · b · r
```

In turn, the above depends upon our providing rules for ConstQ, that is, rules for what will be considered a constant.

```
In[22]:= ConstQ[args_Times] := And @@ ConstQ /@ List @@ args
         ConstQ[args_Plus] := And @@ ConstQ /@ List @@ args
         ConstQ[c_^n_] := ConstQ[c] ∧ ConstQ[n]
         ConstQ[_?NumberQ] := True
         ConstQ[_Symbol] := True
         ConstQ[_] := False
```

Now our calculations work correctly.

```
In[28]:= Ĵ+ · Ĵ- · |j_Ĵ, m_Ĵz⟩
```

$$\text{Out[28]= } (j\ (1+j) - (-1+m)\ m)\ \hbar^2\ \left|j_{\hat{J}},\ m_{\hat{J}_z}\right\rangle$$

In angular momentum theory, the operator $\hat{J}^2$ has the eigenvalue $j(j+1)\hbar^2$ on eigenstates of angular momentum. It is usually a textbook calculation to show that the operator $\hat{J}^2$ can be decomposed into operators involving $\hat{J}_+$, $\hat{J}_-$, and $\hat{J}_z$. Let us verify that this is the case.

```
In[29]:= (1/2 (Ĵ+ · Ĵ- + Ĵ- · Ĵ+) + Ĵz · Ĵz) · |j_Ĵ, m_Ĵz⟩
```

$$\text{Out[29]= } m^2\ \hbar^2\ \left|j_{\hat{J}},\ m_{\hat{J}_z}\right\rangle + \frac{1}{2}\ ((j\ (1+j) - (-1+m)\ m)\ \hbar^2\ \left|j_{\hat{J}},\ m_{\hat{J}_z}\right\rangle + (j\ (1+j) - m\ (1+m))\ \hbar^2\ \left|j_{\hat{J}},\ m_{\hat{J}_z}\right\rangle)$$

```
In[30]:= Simplify @ %
```

$$\text{Out[30]= } j\ (1+j)\ \hbar^2\ \left|j_{\hat{J}},\ m_{\hat{J}_z}\right\rangle$$

For those that know enough about quantized angular momentum, it is evident that the calculation above shows that $\hat{J}^2$ is equivalent to $\frac{1}{2}\ (\hat{J}_+ \cdot \hat{J}_- + \hat{J}_- \cdot \hat{J}_+) + \hat{J}_z \cdot \hat{J}_z$. In fact, we could verify this just by the appropriate substitution of the raising and lowering operators in terms of the operators $\hat{J}_x$ and $\hat{J}_y$.

```
In[31]:= (1/2 Ĵ- · Ĵ+ + 1/2 Ĵ+ · Ĵ- + Ĵz · Ĵz) /. {Ĵ+ → Ĵx + i Ĵy, Ĵ- → Ĵx - i Ĵy} // Expand
```

$$\text{Out[31]= } \hat{J}_z \cdot \hat{J}_z + \hat{J}_x \cdot \hat{J}_x + \hat{J}_y \cdot \hat{J}_y$$

Let us extend our implementation of angular momentum a little further, by adding the additional information that the eigenstates of angular momentum are orthogonal.

```
In[32]:= ⟨j1_Ĵ, m1_Ĵz| · |j2_Ĵ, m2_Ĵz⟩ := δ_{j1,j2} δ_{m1,m2}
```

We can now evaluate the decomposition of $\hat{J}^2$ between different eigenstates of angular momentum.

```
In[33]:= ⟨k_Ĵ, n_Ĵz| · (1/2 (Ĵ+ · Ĵ- + Ĵ- · Ĵ+) + Ĵz · Ĵz) · |j_Ĵ, m_Ĵz⟩
```

```
Out[33]= m^2 ħ^2 δ_{k,j} δ_{n,m} +
          1/2 ((j (1 + j) - (-1 + m) m) ħ^2 δ_{k,j} δ_{n,m} + (j (1 + j) - m (1 + m)) ħ^2 δ_{k,j} δ_{n,m})

In[34]:= Simplify @ %

Out[34]= j (1 + j) ħ^2 δ_{k,j} δ_{n,m}
```

We can calculate expressions involving products of operators between eigenstates.

```
In[35]:= ⟨k_ĵ, n_Ĵz | · Ĵ+ · Ĵz · (Ĵ- + Ĵ+) · |3_ĵ, 0_Ĵz⟩

Out[35]= -12 ħ^3 δ_{k,3} δ_{n,0} + 2 √30 ħ^3 δ_{k,3} δ_{n,2}
```

To reinforce the fact that notations are critically important, note that although the input expression above is very intuitive, at least to physicists, its full form or ASCII form is almost incomprehensible.

```
In[36]:= InputForm[HoldForm[⟨k_ĵ, n_Ĵz | · Ĵ+ · Ĵz · (Ĵ- + Ĵ+) · |3_ĵ, 0_Ĵz⟩]]

Out[36]//InputForm=
HoldForm[NonCommutativeTimes[Bra[EigenLabel[k, Operator[J]],
    EigenLabel[n, Operator[J_Subscript_z]]],
Operator[J_Subscript_Plus],
  Operator[J_Subscript_z], Operator[J_Subscript_Dash] +
Operator[J_Subscript_Plus],
  Ket[EigenLabel[3, Operator[J]], EigenLabel[0,
Operator[J_Subscript_z]]]]]
```

If should now be evident that not having the ability to represent structures in the way a physicist is used to viewing them can be extremely detrimental to understanding and working with these structures.

Before finishing this section, let us reiterate an important point. Although we have used non-commutative times, our approach is limited, and the correct handling of non-commutative operators requires a somewhat different approach. Describing the changes that are necessary to achieve proper handling of such structures is the topic of §4 *Language Modifications* and §5 *Prototypical Structures and Quantum Mechanics*.

2.8 Closing Comments

A major milestone in computer algebra systems has been achieved. Indeed, only 5 years ago Davenport said it would not be possible to have a system where the input and output were substantially the same [87]. Actually, on an historical note, when the author first embarked upon the *Notation* package, some developers *inside* Wolfram Research said that to develop such a package was an impossible task. Happily, they were wrong, and users the world over are the richer for it.

This chapter has introduced the *Notation* package, and has given a wide range of examples from the simple to the complex. I have specifically tried to limit the amount of physics needed in this section, in order to appeal to as wide a range of talents and backgrounds as possible. But the reader should be under no doubts that physicists, amongst all the physical scientists, are most in need of having the ability to create notations in the settings of symbolic computation. For instance, to give an idea of how far one can push the *Notation* package, and indeed to what extent physicists *need* notations, consider the double tensor constructed by the coupling of two spherical tensor operators in quantum mechanics.

$$\mathbf{a}^{\dagger} \cdot \mathbf{a}^{\kappa\_ k\_}_{\rho\_ q\_} := \sum_{\xi,\eta} \langle s, m_{s,\xi}, s, m_{s,\eta} \mid s, s, \kappa, \rho \rangle \langle 1, m_{1,\xi}, 1, m_{1,\eta} \mid 1, 1, k, q \rangle \mathbf{a}^{\dagger}_{\xi} \cdot \tilde{\mathbf{a}}_{\eta} \tag{2.8.a}$$

Setting up these structures and doing computations in an elegant and intuitive way without the *Notation* package would be *extremely* difficult, if not impossible. For instance, in the above we have used a tensorial notation $\mathbf{a}^{\dagger} \cdot \mathbf{a}^{\kappa\_ k\_}_{\rho\_ q\_}$, a notation for abstract sums ($\sum_{\xi,\eta}$), a new notation for correctly assigning tensor values (:≅), a notation for bra-kets, and finally a notation for creation and annihilation operators. If we did not have all these notations, then (2.8.a) would not look like an expression from physics to a physicist. Indeed, we will perform calculations such as the one above. In fact, statements simliar to (2.8.a) occur in §7 *Tensor Calculus, Applications, and Quasi-Spin*. It also appears in almost the exact same form in Wybourne[344]. Thus, the need for diverse notations and their applicability in physics should be apparent.

Where possible, one should follow standard *Mathematica* conventions or follow the conventions of a given field. Inventing one's own individual non-standard notations is discouraged, since such notations are necessarily unrecognizable by other users. Even if a notation has an historical origin and is not as intuitive as other possible notations that one could invent, it is usually better, where possible, to use the historical notation. Admittedly, however, it is sometimes difficult to resolve the inconsistencies present in a certain notation with the desire to have a uniform notation.

One should endeavor not to alter standard *Mathematica* conventions too much. For example, changing commas to vertical separators is strongly discouraged. The more notational oddities present in the system, the higher the chance that one notation will adversely interact with another, giving unexpected results. Some parsers have mechanisms to detect ambiguity or conflicts in a given grammar, but the *Notation* package does not. It effects, in essence, a dynamic grammar with the grammar potentially changing at any stage. Although some work has been done on dynamic grammars [28, 159, 283], certainly none of the standard parser generation tools such as Yacc[20], Lex[20], Bison[76], Flex[256], etc., routinely handle dynamic grammars.

On a related point, some authors — for example Ghezzi & Jazayeri [125], Hoare[162], and Wirth[340] — comment that in some languages, like PL/I and Ada, it is possible to create "dialects". Unfortunately, this means that other people that do not understand the new dialect

may find programs using the dialect indecipherable. This is another strong and compelling argument for why one should not deviate from standard conventions.

The introspective reader may feel that, with the *Notation* package, one can launch a multitude of dialects and hence make interoperability impossible. The refutation of this challenge is that although programmers now have the flexibility to create a vast number of different notations using the *Notation* package, the overriding point is that they can now create *the specific dialect* that is used in their particular field. This can only be a good thing.

Chapter 3
Foundations of *Notation*

3.1 Introduction

In the previous chapter, we showed how to introduce new notations through the use of the three main functions of the *Notation* package — `Notation`, `Symbolize` and `InfixNotation`. We also showed how to use the various options available which are specific to the *Notation* package. We briefly discussed box structures — the underlying representations used by the *Mathematica* front end. This included a brief introduction to tag boxes, which are vital to the working of the *Notation* package. Precedence and grouping of operators were considered, especially in relation to new composite operators which one can define using the *Notation* package. This included material on how one can alter grouping behavior by the use of tag boxes. The chapter culminated with a sophisticated example — that of the bra-ket notation of Quantum Mechanics — using many of the notions introduced up to that point.

The present chapter examines the principles underlying the material of the previous chapter. We will look at how new notation definitions are built up at the box structure level by the *Notation* package. This includes a discussion of how tag boxes work in general and which specific tags are defined by the *Notation* package. We will then describe how to use complex patterns inside notations — for example, how to create a notation that needs some parameter to be an integer. Using this new knowledge, we will then build up a notation for tensors and consider some simple usages of this tensor notation. Finally, we conclude this chapter with a discussion about future possibilities and give some closing remarks.

3.2 Principles behind the Notation Package

In this section, we come to grips with how the *Notation* package works at a fundamental level. We start off, in §3.2.1 *The Functioning behind the Notation Package*, by looking at the parsing and formatting rules produced by a notation statement. How these are able to operate on notations not recognized by the parser depends intimately on how tag boxes function. So, we consider tag boxes in general in §3.2.2 *Tag Boxes*, and the particular tag box `NotationBoxTag` in §3.2.4 *The Tag NotationBoxTag*. We are then finally in a position to illustrate how the compiler of the *Notation* package is able to parse and format new notation.

If not already so, the reader should become familiar with the concepts in §2.8 *Textual Input and Output* of the *Mathematica* Book [342], in particular §2.8.3 *The Representation of Textual Forms*, §2.8.4 *The Interpretation of Textual Forms*, §2.8.10 *Representing Textual Forms by Boxes*, §2.8.12 *String Representation of Boxes*, §2.8.13 *Converting between Strings, Boxes and Expressions*, and finally §2.8.17 *Advanced Topic: Low-Level Input and Output Rules*.

3.2.1 The Functioning behind the Notation Package

The *Notation* package is, in essence, a notation compiler — see Aho [6]. In §2.5.3 *Modifying the Grammar*, we stated that the *Notation* package uses each notation statement to produce some additional rules for `MakeExpression` and/or `MakeBoxes`, which implement the notation. Actually, notation statements produce rules for two new functions : `NotationMakeExpression` and `NotationMakeBoxes`. These functions, introduced by the *Notation* package, behave in *exactly the same* way as their counterparts, `MakeExpression` and `MakeBoxes`. The reason for introducing and using these new functions will be discussed near the end of this section.

Since we are starting this chapter in a fresh *Mathematica* session, as a prelude to our discussions we must load the notation package and the set of common notations we developed in the last chapter. (These common notations are listed in §A.1 *Common Notations*.)

```
In[1]:= << Utilities`Notation`
        << CommonNotations`
```

Let us consider a very simple `Notation` statement, and use the `Action` option to see what rule(s) the statement would generate, *if* we entered it.

```
In[3]:= Notation[x_± ⟺ superPlusMinus[x_], Action → PrintNotationRules]
```

```
NotationMakeExpression[SuperscriptBox[x_, "±"], StandardForm] :=
   MakeExpression[
      RowBox[{"superPlusMinus", "[", x, "]"}], StandardForm]

NotationMakeBoxes[superPlusMinus[x_], StandardForm] :=
   SuperscriptBox[MakeBoxes[x, StandardForm], "±"]
```

Note that these rules are just ordinary assignment statements. (One can easily check that the above `Notation` statement with '$\Longleftrightarrow$' replaced by '$\Longrightarrow$' or '$\Longleftarrow$' would generate just the first rule or second rule, respectively.)

Now consider a simple input, say `(3 + 5)`$^{\pm}$. Its corresponding box structure is

```
SuperscriptBox[RowBox[{
            "(", RowBox[{"3", "+", "5"}], ")"}], "±"]
```
(3.2.a)

For such an input box structure, the `NotationMakeExpression` rule printed above would match and return

```
MakeExpression[RowBox[{
         "superPlusMinus", "[",
         RowBox[{"(", RowBox[{"3", "+", "5"}], ")"}],
         "]"}],
      StandardForm]
```
(3.2.b)

This in turn results in the kernel expression

```
HoldComplete[superPlusMinus[Plus[3, 5]]]
```
(3.2.c)

The above passes to the evaluator, where the `HoldComplete` is stripped off and the result evaluated to `superPlusMinus[8]`. The rules for `NotationMakeBoxes` are then applied, resulting in

```
SuperscriptBox[MakeBoxes["8", StandardForm], "±"]
```
(3.2.d)

which is sent to the front end, where it is displayed as the textual output $8^{\pm}$.

Having seen the above, the reader should now have a better understanding of how an input expression containing some new notation is converted to a corresponding expression free of this new notation, which is in turn evaluated, and then the result appears as an output expression (possibly involving the new notation). For more complex notations, the rules generated for `NotationMakeExpression` and `NotationMakeBoxes` are considerably more complex, as the following demonstrates.

```
In[4]:= Notation[x_ +𝒢 y_ ⟺ GroupPlus[x_, y_], Action → PrintNotationRules]

NotationMakeExpression[RowBox[{Utilities`Notation`Private`l___,
            x_, SubscriptBox["+", "𝒢"], y_,
            Utilities`Notation`Private`r___}], StandardForm] :=
   MakeExpression[RowBox[{Utilities`Notation`Private`l,
            RowBox[{"GroupPlus", "[", RowBox[{x, ",", y}], "]"}],
            Utilities`Notation`Private`r}], StandardForm]
```

```
NotationMakeBoxes[x_ +𝒢 y_, StandardForm] := RowBox[
      {Parenthesize[x, StandardForm, Plus], SubscriptBox["+", "𝒢"],
         Parenthesize[y, StandardForm, Plus]}]
```

There is an aspect of what we have just seen which is somewhat subtle, but is nonetheless important if one wants to understand how to construct notation statements involving complex patterns, to be discussed in §3.3.1 *Complex Patterns and Notation Pattern Tags I.* What we have just seen is that when making use of a `Notation` statement, pattern matching on the external representation is performed on box structures while pattern matching on the internal representation is performed on full form expressions. In our example, it is the box structure of $(3 + 5)^{\pm}$, that is (3.2.a), which matches the box structure `SuperscriptBox[x_, "±"]` appearing in the `NotationMakeExpression` rule; and it is the full form version `superPlusMinus[8]` of $8^{\pm}$ which matches the full form expression `superPlusMinus[x_]` appearing in the `NotationMakeBoxes` rule.

Let us briefly discuss why the *Notation* package has used alternatives for `MakeExpression` and `MakeBoxes`. The functions `NotationMakeExpression` and `NotationMakeBoxes` perform almost exactly the same functions as their counterparts; but they have been used so that the *Notation* package has a minimal interaction or overlap with other packages that use or modify `MakeExpression` and `MakeBoxes`. Also, using these alternative functions allows a much cleaner separation between the rules a user might define and the machine generated rules of the *Notation* package.

What is still not clear, even after the above example, is why the `Notation` statement for $x_{\pm}$, which is used by the *Notation* compiler to generate the above rules, is itself admissible to *Mathematica*'s parser. For that statement contains a subexpression, namely $x_{\pm}$, which is not yet parsable by *Mathematica,* hence the whole `Notation` statement would seem to be not yet parsable. Simply stated, how does the notation creation process get started? The answer: by making appropriate use of tag boxes.

As we saw in §2.5.4 *Tag Boxes in Notation Statements,* each of the placeholders in the box structure of the template `Notation[□ ⟺ □]` is encapsulated by a hidden tag box with the tag `NotationBoxTag`. This is also true of each '□' appearing in the templates for `Notation[□ ⟹ □]`, `Notation[□ ⟸ □]`, `Symbolize[□]`, `InfixNotation[□, □]`, and `AddInputAlias[□, □]`. In the next two subsections, we will elucidate how tag boxes work in general, and in particular how they work when they are tagged by `NotationBoxTag`.

3.2.2 Tag Boxes

To illustrate tag boxes, consider the following input.

$$x^2 \qquad (3.2.e)$$

Looking at the above, one would probably guess that its box structure is `SuperscriptBox["x", "2"]`. Actually, it is

```
TagBox[SuperscriptBox["x", "2"], foo]
```
(3.2.f)

as one can check by using **Show Expression**. Tag boxes are visually hidden from the user.

The default parsing of a box of the form `TagBox[`*boxes*, *tag*`]` is *tag*`[`*expr*`]`, where *expr* is what *boxes* is parsed to. For example,

```
In[5]:= a + b                    Box structure is TagBox[RowBox[{"a", "+", "b"}], foo]
Out[5]= foo[a + b]
```

However, in the case when we enter a currently inadmissible notation like $expr^{\pm}$, even though it is surrounded by a hidden tag box with the tag `foo`, as in the following example, it is still inadmissible.

```
In[6]:= x±                       Box structure is TagBox[SuperscriptBox["x", "±"], foo]
      Syntax::sntxi : Incomplete expression; more input is needed.
      x±
```

The example above, demonstrating an inability to parse tag boxes containing unparsable parts, would not necessarily fail if the tag `foo` had a special parsing behavior associated with it. We can define our own rules for the way specific tag boxes are parsed. For instance, by using the low level function `MakeExpression`, the following rule will change how *Mathematica* parses expressions containing a `TagBox` with the tag `literalBoxes`. (Remember, a `MakeExpression` statement must evaluate to a `HoldComplete`, to stop premature evaluation.)

```
In[6]:= MakeExpression[TagBox[boxes_, literalBoxes], form_] :=
          HoldComplete[boxes]
```

The tag `literalBoxes` now has the special parsing behavior associated with it of creating an expression consisting of the very same box structure encapsulated by the tag box wrapper. It is important to note that this works, *whether or not the box structure encapsulated is normally acceptable*. This is illustrated by the following list, neither of whose members is normally acceptable, and each member having a hidden `literalBoxes` tag box surrounding it.

```
In[7]:= {3!, x±}
Out[7]= {SubscriptBox["3", "!"], SuperscriptBox["x", "±"]}
```

Note also that all this is happening quite independently of the *Notation* package.

Because of the new `MakeExpression` statement, any input has become admissible if wrapped in a hidden `literalBoxes` tag box. Such input could be an entire notation statement, so already we see at least one way of making a notation statement admissible, even though it contains parts which are ordinarily inadmissible.

To see in detail what happens, consider for an example the above $x^{\pm}$, which has the box form

```
TagBox[SuperscriptBox["x", "±"], literalBoxes]
```
(3.2.g)

This is sent to the kernel to be converted to an expression by the `MakeExpression` rules. None of the built-in `MakeExpression` rules apply, so ordinarily this box structure could not be parsed and an error message would ensue. However, there is now a `MakeExpression` rule to handle this tag box structure, namely the one defined above involving `literalBoxes`. Using this new rule, the kernel parses the input box structure (3.2.g) as follows.

```
In[8]:= MakeExpression[TagBox[SuperscriptBox["x", "±"], literalBoxes]]

Out[8]= SuperscriptBox["x", "±"]hc
```

Evaluation then takes place, giving `SuperscriptBox["x", "±"]`. This, in turn, is formatted by the kernel using the built in rules for `MakeBoxes`.

```
In[9]:= MakeBoxes[SuperscriptBox["x", "±"]]

Out[9]= RowBox[{"SuperscriptBox", "[", RowBox[{"\"x\"", ",", "\"±\""}], "]"}]
```

Finally, this result is sent to the front end, where it is displayed as follows.

```
In[10]:= DisplayForm @ %

Out[10]//DisplayForm=
        SuperscriptBox["x", "±"]
```

We now have some idea of how tag boxes work in general and how special behaviors can be set up for specific tags. We are at last in a position to learn how the special tags defined by the *Notation* package actually work. There are three such special tags: `NotationBoxTag`, `NotationPatternTag` and `NotationMadeBoxesTag`. We examine `NotationBoxTag` next, and subsequently examine the other tags in §3.3.1 *Complex Patterns and Notation Pattern Tags* and §3.3.5 *NotationMadeBoxesTag*.

3.2.3 Tags in Tag Boxes

In some previous subsections, such as §2.6.5 *Changing Grouping Behavior* and §2.7.1 *Prototypical Ket Structures*, the tags we used in the tag boxes were strings, *not* symbols. Let us now give a fuller explanation of why this choice was made. The easiest way to demonstrate the potential problems arising from the use of symbols, as opposed to strings, as tags in tag boxes is to give an example of creating a notation inside a small package. First, let us clear all of the notations currently active.

```
In[11]:= ClearNotations[]
```

Next, inside a dummy package, let us recreate a variant of the notation for our interior product, which we gave in §2.6.6 *Changing Precedences and the Option SyntaxForm*. We modify the prototypical interior product tag box structure, (2.6.n), to use a symbol as opposed to a tag, resulting in the prototypical structure (3.2.h).

```
TagBox[StyleBox["⌋",
      AutoStyleOptions → {"UnmatchedBracketStyle" → None}],
   InteriorProductTag, SyntaxForm → "*",
   Editable → False, Selectable → False]
```
(3.2.h)

Here is the code which starts the dummy package.

```
In[12]:= BeginPackage["DummyPackage`", "Utilities`Notation`"];
         Begin["`Private`"];
```

The following notation statement, based on (3.2.h), creates the interior product notation.

```
In[14]:= Notation[v_ ⌋ ω_ ⟺ InteriorProduct[v_, ω_]];
```

Finally, the package is ended by the following.

```
In[15]:= End[];
         EndPackage[];
```

But the notation inside the package does not work outside the package. (This might be considered a good thing.)

```
In[17]:= v_ ⌋ (ω_ + μ_) := v ⌋ ω + v ⌋ μ

         Syntax::sntxf : "v_" cannot be followed by "⌋ (ω_ + μ_)".

         v_ ⌋ (ω_ + μ_) := v ⌋ ω + v ⌋ μ
```

To see why this is so, we must examine the rules the *Notation* package created. Instead of performing the query `??NotationMakeExpression`, we only quote here, due to space considerations, the single produced rule.

```
NotationMakeExpression[
      RowBox[{Utilities`Notation`Private`l___,
            DummyPackage`Private`v_,
            TagBox[StyleBox["⌋", AutoStyleOptions →
                  {"UnmatchedBracketStyle" → None}],
               DummyPackage`Private`InteriorProductTag,
               Editable → False, Selectable → False,
               SyntaxForm → "*"], DummyPackage`Private`ω_,
            Utilities`Notation`Private`r___}],
      "StandardForm"] := MakeExpression[
      RowBox[{Utilities`Notation`Private`l,
            RowBox[{"InteriorProduct", "[",
                  RowBox[{DummyPackage`Private`v, ",",
                        DummyPackage`Private`ω}], "]"}],
            Utilities`Notation`Private`r}], "StandardForm"]
```
(3.2.i)

Examining (3.2.i), we see that instead of creating a pattern which matches a tag box of the form

```
TagBox[…, InteriorProductTag]
```

it has instead created a pattern which matches

```
TagBox[…, DummyPackage`Private`InteriorProductTag]
```

Thus, unless we change the tags in our input, we will not get a match. There are two solutions: either declare the symbol `InteriorProductTag` in the public context of the package, or use a string tag. If we use a symbol, then we must create a usage string for the symbol, for example "InteriorProductTag is a symbol used in the parsing of interior products." Also, if the user accidentally enters a statement that uses a notation in some package not yet loaded, then the user can unfortunately load that symbol into the global name space. There will then be issues surrounding the shadowing of symbols, when the package is loaded. By using a string tag, we avoid any issues of the context of the symbol and avoid worrying about which context the notation was declared in.

String tags were not used in the previous subsection, §3.2.2 *Tag Boxes*. This was done for the sake of simplicity since we were only explaining the functionality of tag boxes. Unfortunately, they were not used in the *Notation* package since, at the time of creating the package, the author was not aware that string tags were allowed. The only excuse for this is perhaps that tag boxes have almost no documentation in the *Mathematica* Book [342], or elsewhere to my knowledge. Thus, when discerning the functionality of tag boxes, the author just used examples occurring within *Mathematica*. Only at a later date was it realized that string tags were in fact permissible. For backward compatibility, the *Notation* package still uses tags which are symbols.

Although not critical, it is recommended that string tags are used whenever possible.

3.2.4 The Tag NotationBoxTag

Upon loading, the *Notation* package sets up rules for how the tag `NotationBoxTag` is handled in input and output. It is this tag which allows us to capture the box structures of notations when the notations are entered into *Mathematica*. This tag behaves somewhat like `literalBoxes`, considered in the previous subsection, allowing normally unparsable expressions to be parsable. Tag boxes containing this tag are embedded in every `Notation`, `Symbolize`, `InfixNotation`, and `AddInputAlias` statement in *Mathematica*.

`TagBox` *tag name*	*visual form*	*effect of tag*
`NotationBoxTag`	*...expression...*	allows notation statements to parse notations which are not yet admissible

The visual form and behavior of the tag NotationBoxTag.

In the remainder of this section, let us assume that the private context of the *Notation* package has been added to the context path, so that our discussion is more readable. (This would easily be accomplished by something like `PrependTo[ $ContextPath, "Utilities`Notation`Private`"]`.) Under this assumption, the rules involving `NotationBoxTag` set up by the *Notation* package look like the following.

```
NotationMakeExpression[TagBox[boxes_, NotationBoxTag, opts___],
    anyForm_] := HoldComplete[NotationBoxTag[boxes]]
```
(3.2.j)

```
NotationMakeBoxes[NotationBoxTag[boxes__], anyForm_] :=
   TagBox[boxes, NotationBoxTag]
```
(3.2.k)

Let us focus on the above NotationMakeExpression rule. It involves the tag NotationBoxTag and is rather similar to the MakeExpression rule in the previous section which involved the tag literalBoxes. However, the parsed expression of a NotationBoxTag tag box will contain a wrapper. For example,

```
In[17]:= NotationMakeExpression[ TagBox[SuperscriptBox["x", "±"],
              NotationBoxTag], StandardForm] // FullForm
Out[17]//FullForm=
      HoldComplete[NotationBoxTag[SuperscriptBox["x", "±"]]]
```

We can now illustrate how the compilation engine of the *Notation* package works, at least for simple examples. Consider a notation statement of the form

```
Notation[leftExpr ⟺ rightExpr]
```
(3.2.l)

Assume *leftBoxes* is the box structure corresponding to *leftExpr*, hence has a hidden surrounding NotationBoxTag tag box, and similarly for *rightBoxes* and *rightExpr*. The box structure of the notation statement would essentially be

```
RowBox[{"Notation", "[", RowBox[{
    TagBox[ leftBoxes, NotationBoxTag],
            " ", "⟺", " ",
    TagBox[rightBoxes, NotationBoxTag]}], "]"}]
```
(3.2.m)

Because of the specific way NotationBoxTag is handled, as dictated by (3.2.j), TagBox[*leftBoxes*, NotationBoxTag] is parsed to NotationBoxTag[*leftBoxes*], and similarly for *rightBoxes*. Thus overall, the notation statement is parsed to

```
HoldComplete[Notation[DoubleLongLeftRightArrow[
        NotationBoxTag[leftBoxes], NotationBoxTag[rightBoxes]]]]
```
(3.2.n)

where *leftBoxes* and *rightBoxes* appear literally, that is, unparsed. To see a specific case of the above, let us consider the notation statement for $x^{\pm}$ treated previously in §3.2.1 *The Functioning behind the Notation Package*. The following shows what it is parsed to.

```
In[18]:= Hold[Notation[x_^± ⟺ superPlusMinus[x_]]] // FullForm
Out[18]//FullForm=
      Hold[Notation[DoubleLongLeftRightArrow[
            NotationBoxTag[SuperscriptBox["x_", "±"]], NotationBoxTag[
              RowBox[List["superPlusMinus", "[", "x_", "]"]]]]]]]
```

The *Notation* compiler then uses this statement (with the 'Hold' removed) to generate the specific NotationMakeExpression and NotationMakeBoxes rules shown in §3.2.1 *The Functioning behind the Notation Package*. Returning to our generic example, the *Notation* package would, by a rather sophisticated algorithm, generate new rules of the form

```
NotationMakeExpression[processedLeftBoxes, StandardForm] :=
    MakeExpression[processedRightBoxes, StandardForm]
NotationMakeBoxes[processedRightBoxes, StandardForm] :=
    heavilyProcessedLeftBoxes
```
(3.2.o)

where the *processedBoxes* are all different.

To describe further the inner workings of the *Notation* package in general, or even just the algorithms which generate the `NotationMakeExpression` and the `NotationMakeBoxes` rules, would embroil us in rather technical considerations. These details are beyond the focus of this thesis. Moreover, besides embodying some sporadic and useful ideas, much of the code is not that insightful. The interested reader, however, can consult the source code of the *Notation* package. It is located in the normal *Mathematica* distribution in the directory math/AddOns/ExtraPackages/Utilities/Notation/Documentation/English/(at least for versions 3.0 and 4.0).

3.3 Complex Patterns in Notations

Up to this point, the only patterns that have appeared in notation statements have been *simple* ones, that is, patterns of the form *label_*, *label__*, or *label___*. In §3.3.1 *Complex Patterns and Notation Pattern Tags*, we describe how to include conditional patterns and complex patterns in notation statements. Conditional patterns are sometimes necessary to ensure that a notation only works for say integers or maybe to ensure that a notation conforms to a certain form — for instance, in tensors two indices cannot appear directly above and below each other. In addition, we introduce the tag `NotationPatternTag`, to ensure that what appears to be a conditional or complex pattern is so interpreted. There are, however, times when we want complex patterns to be interpreted literally, and one such case is discussed in §3.3.3 *The Need for Literal Patterns*. In §3.3.5 *NotationMadeBoxesTag*, we expose the reasons why we sometimes do not want an expression appearing in a `Notation` statement to be processed any further, and we introduce the tag `NotationMadeBoxes` to accomplish this.

3.3.1 Complex Patterns and Notation Pattern Tags I

For normal purposes, it is usually sufficient that the patterns present in `Notation` and `Symbolize` statements are simple patterns. However, it is sometimes necessary or desirable to use more complicated patterns in notations. For example, we might want a notation to be valid only when a certain pattern is matched by a number.

To introduce complex patterns inside a notation statement, we must surround each such pattern by a tag box with the tag `NotationPatternTag`. Then, when the *Notation* package is generating the rules which embody the notation statement being entered, it can act accordingly. This wrapping is critical, since if what appears to be a complex pattern is not embedded in such a tag box, then it will be treated as a literal string of characters to be matched and not function as a pattern.

TagBox *tag name*	*visual form*	*effect of tag*
`NotationPatternTag`	... *subexpression* ...	forces the contents of the surrounded structure to be treated as an expression rather than as a literal structure

The visual form and behavior of the tag NotationPatternTag.

To illustrate, consider the following notation statement, which has no embedded `NotationPatternTag` tag box.

```
In[1]:= Notation[∇[any_?NumericQ] ⟺ goo[any_]]
```

Although one might think that the following input should now be recognizable to *Mathematica*, it isn't.

```
In[2]:= ∇[3]

        Syntax::sntxf : "∇" cannot be followed by "[3]".

        ∇[3]
```

In contrast, the following is parsable.

```
In[2]:= ∇[y?NumericQ] // FullForm
Out[2]//FullForm=
        goo[y]
```

From these two examples, it is evident that the notation has been defined to match literally something of the form `∇[` *something* `?NumericQ]`. Thus '`?NumericQ`' is being treated as a string of characters which must be literally present in order to effect a match. This is exactly analogous to the fact, discussed in §2.2.4 *Notation: Explicit Bracketing*, that if brackets are involved in declaring a notation, then they must also be present in any input that is intended to use the notation.

To illustrate complex patterns, let us define a notation which accepts input of the form $\Gamma_{num\_}$ only when the subscript expression *num* is a numerical expression/value. As a preliminary, recall from §3.2.1 *The Functioning behind the Notation Package*, that pattern matching on an external representation is actually performed on box structures, not expressions. Consequently, we usually have to create small variants of our testing functions in order to use them inside complex patterns. In practice, this amounts to defining box handling functions similar to the corresponding expression handling functions. For instance, for our present example, we must define the function `unparsedNumericQ`, which is analogous to `NumericQ` but operates on unparsed box structures.

```
In[3]:= unparsedNumericQ [ boxes___ ] := NumericQ [ ToExpression [ boxes] ]
```

We are now in a position to create the notation which only recognizes input of the form Γ_{num}, where *num* must parse as a number. We do this by the following.

```
In[4]:= Notation [Γnum_?unparsedNumericQ ⟺ foo [num_?NumericQ] ]
```

We use `NumericQ` rather than `unparsedNumericQ` on the right hand side of the statement, since pattern matching on an internal representation *does* follow conventional pattern matching. Any substructure surrounded by a tag box with the tag `NotationPatternTag` as well as having it's `TagStyle` set to `"NotationPatternWrapperStyle"` is visually distinguished by having a pink background, like *subexpression*.

One would enter the previous notation statement by first entering a `Notation[□ ⟺ □]` template, then replacing the '□'s with the external and internal forms as seen above, and then finally selecting the subexpressions *num*`_?unparsedNumericQ` and *num*`_?NumericQ` in succession, each time clicking the button **InsertPatternWrapper** (which is located in the **Wrapper Boxes** part of the full notation palette). Each such click wraps a tag box (with the tag `NotationPatternTag`) around what is selected and tints the background of the selection to indicate that a complex pattern is present.

The following shows that our `Notation` statement works as intended. In particular, only input of the form Γ_{num}, where *num* can be considered a numerical expression, will be interpreted as a `foo` object.

```
In[5]:= Γπ + Γh // FullForm
Out[5]//FullForm=
        Plus[foo[Pi], Subscript[Γ, h]]
```

Reciprocally, only `foo` objects with numerical arguments will be formatted using the new notation.

```
In[6]:= foo[π] + foo[h]
Out[6]= Γπ + foo[h]
```

It is highly instructive to examine the form of the rules created by our `Notation` statement.

```
In[7]:= Notation [Γnum_?unparsedNumericQ ⟺ foo [num_?NumericQ],
           Action → PrintNotationRules]

        NotationMakeExpression[
              SubscriptBox["Γ", num_?unparsedNumericQ], StandardForm] :=
           MakeExpression[RowBox[{"foo", "[", num, "]"}], StandardForm]

        NotationMakeBoxes[foo[num_?NumericQ], StandardForm] := SubscriptBox[
              "Γ", Utilities`Notation`Private`makeEvaluatedRowBoxOfBoxes [
                 {num}, StandardForm, None]]
```

Observe that when it comes to parsing, *num*`_?unparsedNumericQ` appears in the `NotationMakeExpression` statement as a pattern, not as a literal string, since in the notation statement it is surrounded by a `NotationPatternTag`. This is as intended. For instance, `Sub-`

`scriptBox["Γ", "3"]` (the box structure of Γ_3) matches `SubscriptBox["Γ", num_?unparsedNumericQ]` (the pattern in the `NotationMakeExpression` rule). Note also the importance of our having used `unparsedNumericQ` rather than `NumericQ`, since `SubscriptBox["Γ", "3"]` would not match `SubscriptBox["Γ", num_?NumericQ]`. Similarly, when it comes to formatting, *num*_`?NumericQ` appears in the `NotationMakeBoxes` statement as a pattern, not as a literal string, since it too is surrounded by a `NotationPatternTag`.

For completeness, we give the underlying structure of the notation statement.

```
RowBox[{"Notation", "[",
      RowBox[{
        TagBox[
          SubscriptBox["Γ",
            TagBox[
              RowBox[{"num_", "?", "unparsedNumericQ"}],
              NotationPatternTag,
              TagStyle->"NotationPatternWrapperStyle"]],
          NotationBoxTag,
          TagStyle->"NotationTemplateStyle"], " ", "⟺", " ",
        TagBox[
          RowBox[{"foo", "[",
            TagBox[
              RowBox[{"num_", "?", "NumericQ"}],
              NotationPatternTag,
              TagStyle->"NotationPatternWrapperStyle"], "]"}],
          NotationBoxTag,
          TagStyle->"NotationTemplateStyle"]}], "]"}]
```
(3.3.a)

3.3.2 Complex Patterns and Notation Pattern Tags II

An application of the general ideas presented in the previous subsection is that of symbolizing expressions subscripted by integers. This is a question that is frequently asked of the author. The problem is the following: if one symbolizes some variables and then creates them in an expression as a by-product of evaluation, then they will not be treated as symbols. For instance, consider the following list.

```
In[8]:= Table[yi, {i, 1, 5}]

Out[8]= {y1, y2, y3, y4, y5}

In[9]:= FullForm @ %

Out[9]//FullForm=
List[Subscript[y, 1], Subscript[y, 2],
   Subscript[y, 3], Subscript[y, 4], Subscript[y, 5]]
```

We can see that all the items in the list are expressions, not symbols. Even if we had directly symbolized the y_1, y_2, etc., it would make no difference because symbolization occurs only during parsing, yet the subscripted symbols are created during evaluation. The problem should

now be clear. The solution should probably be equally clear. We simply symbolize objects which fit the complex pattern of $symb_{integer}$. We can do this as follows.

```
In[10]:= Symbolize[ (_?unparsedSymbolQ)_?unparsedNumericQ ]
```

Technical Note: The parentheses in the above symbolization, that is in `(_?unparsedSymbolQ)`, do not affect the pattern since they are part of the complex pattern. Therefore, `(_?unparsedSymbolQ)` is treated as an expression and not as something to be literally matched. Thus the parentheses do not need to be explicitly present in any input that is intended to match the symbolization. (They were included to allow the correct grouping of the input.)

It only remains to define the suitable testing function for `unparsedSymbolQ`.

```
In[11]:= unparsedSymbolQ @ boxes___ := SymbolQ @ ToExpression @ boxes;
         SymbolQ @ _Symbol = True;
         SymbolQ @ ___ = False;
```

Finally, if during evaluation we create a symbol subscripted by an integer, then we can simply transform this to the same expression that parsing the boxes would create.

```
In[14]:= (symb_?SymbolQ)_?IntegerQ := ToExpression @ MakeBoxes @ symb_
```

Observe that now, even subscripted variables created during evaluation are treated as symbols in their own right.

```
In[15]:= Table[y_i, {i, 1, 5}]

Out[15]= {y1, y2, y3, y4, y5}

In[16]:= FullForm @ %

Out[16]//FullForm=
         List[y_Subscript_1, y_Subscript_2,
            y_Subscript_3, y_Subscript_4, y_Subscript_5]
```

Despite complex patterns providing advanced capabilities, it would be remiss not to give the following warning. One should be very careful to avoid unwanted evaluation through testing functions when parsing expressions. For example, when using the notation defined above for Γ_{num}, parsing causes the testing function `unparsedNumericQ` to evaluate its arguments; and as we next see, this can lead to unusual results.

```
In[17]:= Hold[ Γ_Print["Oops..."] ]

         "Oops..."

         "Oops..."

Out[17]= Hold[Γ_Print["Oops..."]]
```

We rectify the parsing of Γ_{num} by changing our testing function, `unparsedNumericQ`, in order to ensure that during testing its argument is not actually evaluated. This can be achieved in various different ways — see any of [123], [222], [325], [293], or specifically [323]. Here is one way which suffices for illustration.

```
In[18]:= unparsedNumericQ @ boxes___ :=
           ReleaseHold [ NumericQ /@
                 Unevaluated /@ ToExpression[boxes, StandardForm, Hold] ]
```

Technical Note: There are many ways we could have coded `unparsedNumericQ`. The above way is probably the simplest.

Now the testing function does not allow any side effect evaluation.

```
In[19]:= Hold[ Γ_Print["Oops..."] ]

Out[19]= Hold[Γ_Print["Oops..."]]
```

Moreover, our notation is still correctly recognized.

```
In[20]:= Γ3 // FullForm

Out[20]//FullForm=
         foo[3]
```

Finally, in accordance with correctly guarding our testing functions against unwanted evaluation, we should rectify the definition of `unparsedSymbolQ` to the following.

```
In[21]:= unparsedSymbolQ @ boxes___ :=
           ReleaseHold [ SymbolQ /@
                 Unevaluated /@ ToExpression[boxes, StandardForm, Hold] ]
```

3.3.3 The Need for Literal Patterns

Given the information above on how complex patterns are incorporated into notations, an obvious question arises. Why are notations with an embedded '?' not interpreted as complex by default? The answer is that it is sometimes necessary to treat the constituents of complex patterns literally. Thus, it transpires that we need to use patterns in dual ways, literally and as pattern expressions. This necessitates a choice. In notation statements, should the default interpretation of patterns be literally as box structures or as pattern expressions? The author's decision was that it is easier and more uniform to adopt the following convention. Unless explicitly indicated otherwise, *everything is to be treated literally except the simple patterns*, that is, `_`, `__`, `___`.

Further justification for the above decision is forthcoming in the next subsection. But for now, let us give a practical example of when we would want to have a complex pattern treated literally. This will involve a simple notation, `?!`, for the opposite of `PatternTest`. Whereas an expression *expr* matches *patt*`?`*test* if *expr* matches *patt* and satisfies *test*, we want *expr* to match *patt*`?!`*test* if *expr* matches *patt* and fails *test*. First, we introduce a notation and a semantics for negating a function.

```
In[22]:= Notation[!f testFunction_ ⟺ notFunction[testFunction_]]

In[23]:= !f test_ = ! test[#] &;
```

Now we define our negated pattern tests in terms of this notation.

```
In[24]:= Notation[patt_?~test_ ⟹ patt_?(!f test_)]

In[25]:= Notation[patt_?!test_ ⟺ patt_?(!f test_)]

         Notation::notationalPatternsUsed :
            Warning: The pattern patt_?(!f test_) is being interpreted
                  as a notation and not a pattern. Use an embedded
                  NotationPatternTag TagBox wrapper if you want
                  this pattern to be treated as a genuine pattern.
```

The warning tells us that as intended, in the `Notation` statement, *patt_*`?(!`$_f$ *test_*`)` is actually being interpreted literally, hence as a notation — a notation which happens to have the *form* of a "pattern test" — rather than *acting* like a "pattern test" which would put restrictions on *patt_*. Let us create a trivial function, `Atomize`, which uses the new notation.

```
In[26]:= Atomize[expr_?!AtomQ] := furtherProcessing[expr];
         Atomize[expr_] := expr
```

Here are a few examples validating the functioning of `Atomize`.

```
In[28]:= Atomize[x]

Out[28]= x

In[29]:= Atomize[x + y]

Out[29]= furtherProcessing[x + y]
```

On a more general note, by looking at the underlying rules that the notation statement generates, we can see that the pattern test is being matched literally. (We turn off the warning first)

```
In[30]:= Off[Notation::notationalPatternsUsed];

In[31]:= Notation[patt_?!test_ ⟺ patt_?(!f test_), Action → PrintNotationRules]

         NotationMakeExpression[RowBox[{Utilities`Notation`Private`l___,
                     patt_, SuperscriptBox["?", "!"], test_,
                     Utilities`Notation`Private`r___}], StandardForm] :=
            MakeExpression[RowBox[{Utilities`Notation`Private`l,
                     RowBox[{patt, "?", RowBox[{"(", RowBox[
                                    {SubscriptBox["!", "f"], test}], ")"}]}],
                     Utilities`Notation`Private`r}], StandardForm]

         NotationMakeBoxes[HoldPattern[PatternTest][patt_, HoldPattern[!f test_]],
               StandardForm] :=
            RowBox[{Parenthesize[patt, StandardForm, PatternTest],
                  SuperscriptBox["?", "!"],
                  Parenthesize[test, StandardForm, PatternTest]}]
```

Specifically, we see that the `NotationMakeBoxes` rule contains

```
HoldPattern[PatternTest][patt, HoldPattern[! (test[#1]) &]]        (3.3.b)
```

This means that when a kernel expression is being converted to boxes, the pattern tests appearing in the kernel expression will be converted according to the above notation, as opposed to the pattern tests in (3.3.b) placing conditions upon which kernel expressions can be converted

3.3.4 Expressions within Notations

The direct motivation for the introduction of the `NotationPatternTag` is to allow complex patterns in a notation statement to function as normal patterns. Nevertheless, these tags allow any piece of a notation statement to be treated as an expression, as opposed to a literal structure. Hence, they have wider application than just too complex patterns. Since this subsection and the next are going to examine many notation statements, let us set the option `Action` to `PrintNotationRules` and then later reset it. Thus, instead of actually entering notation statements, rather we just examine the underlying rules that would be generated if the statements were actually entered.

```
In[32]:= SetOptions[Notation, Action → PrintNotationRules]

Out[32]= {WorkingForm → Automatic, Action → PrintNotationRules}
```

Also, as in previous examples, the function names `foo`, `omega`, etc., have no special significance and are only used to illustrate the form of the notation statements.

In this subsection, we will examine notation statements in pairs. These pairs will only differ in that the first will have a single embedded tag box with a `NotationPatternTag` tag, while the second will have no embedded pattern tags. To fully illustrate the effects of including versus not including such a tag, we will examine notations which parse and notations which format separately.

We start by examining the effects of embedding a `NotationPatternTag` tag box in the *right* hand sides of `Notation` statements. In the case of notations which parse, consider the following.

```
In[33]:= Notation[Ω_any_ ⟹ omega[foo[any_]]]

         NotationMakeExpression[SubscriptBox["Ω", any_], StandardForm] :=
           MakeExpression[RowBox[{"omega", "[", foo[any], "]"}], StandardForm]

In[34]:= Notation[Ω_any_ ⟹ omega[foo[any_]]]

         NotationMakeExpression[SubscriptBox["Ω", any_], StandardForm] :=
           MakeExpression[
             RowBox[{"omega", "[", RowBox[{"foo", "[", any, "]"}], "]"}],
             StandardForm]
```

Observe that in the first statement above, the one with the embedded notation pattern tag, `foo[any]` appears as an actual expression, as opposed to its occurrence in the second statement as a literal structure — `RowBox[ { "foo", "[", any, "]" } ]`. In contrast, in the case of

notations which format, we see below that any notation pattern tags on the right hand side are ignored.

```
In[35]:= Notation[Ω_any_ ⟸ omega[foo[any_]]]

         NotationMakeBoxes[omega[foo[any_]], StandardForm] :=
           SubscriptBox["Ω", MakeBoxes[any, StandardForm]]

In[36]:= Notation[Ω_any_ ⟸ omega[foo[any_]]]

         NotationMakeBoxes[omega[foo[any_]], StandardForm] :=
           SubscriptBox["Ω", MakeBoxes[any, StandardForm]]
```

What is the use of such parsing behavior? We will see a direct application in §3.4 *Tensorial Notation*. But it should be readily apparent that if the particular notation is designed for situations where the box structure of *any* is typically extremely complex, then we could create some sophisticated rules for `foo` and have it interpret the box structure appropriately.

Let us next examine the reciprocal situation, where a notation pattern tag occurs on the other side, the *left* hand side, of a `Notation` statement. We use a small variation of the above notation statements.

```
In[37]:= Notation[Ω_foo[any_] ⟹ omega[any_]]

         NotationMakeExpression[SubscriptBox["Ω", foo[any_]], StandardForm] :=
           MakeExpression[RowBox[{"omega", "[", any, "]"}], StandardForm]

In[38]:= Notation[Ω_foo[any_] ⟹ omega[any_]]

         NotationMakeExpression[SubscriptBox["Ω",
                 RowBox[{"foo", "[", any_, "]"}]], StandardForm] :=
           MakeExpression[RowBox[{"omega", "[", any, "]"}], StandardForm]
```

Observe that in the first statement above, again the one with the embedded notation pattern tag, `foo[any]` appears as an actual expression, as opposed to its occurrence in the second statement as a literal structure — `RowBox[{ "foo", "[", any, "]" }]`. In the formatting case we have the following.

```
In[39]:= Notation[Ω_foo[any_] ⟸ omega[any_]]

         NotationMakeBoxes[omega[any_], StandardForm] := SubscriptBox["Ω",
               Utilities`Notation`Private`makeEvaluatedRowBoxOfBoxes[
                 {foo[any]}, StandardForm, None]]

In[40]:= Notation[Ω_foo[any_] ⟸ omega[any_]]

         NotationMakeBoxes[omega[any_], StandardForm] := SubscriptBox["Ω",
               RowBox[{"foo", "[", MakeBoxes[any, StandardForm], "]"}]]
```

Thus, when formatting, the substructures on the left hand side that are wrapped in a `NotationPatternTag` tag box are evaluated and then made into boxes. Overall, the various combinations are summarized in the following table.

notation statement form	`NotationPatternTag` *behavior*
... ⟹ ... (*struct*) ...	the surrounded substructure is treated like an expression
... ⟸ ... (*struct*) ...	the surrounded substructure is treated as normal
...(*struct*) ... ⟹ ...	the surrounded substructure is treated like an expression
...(*struct*) ... ⟸ ...	the surrounded substructure is treated like an expression, then made into boxes.

The behavior of the tag NotationPatternTag inside notation statements.

The behavior of the notation pattern tag can be extremely useful, especially when we need to have certain bits of code parse dynamically. We will encounter this in the next section, §3.4 *Tensorial Notation*. In summary, complex pattern tags will play a pivotal role; but before this, we must introduce an additional tag that affects how notation statements are generated.

3.3.5 NotationMadeBoxesTag

The tag `NotationMadeBoxesTag` is intended for advanced users. It is used to indicate that box processing and formatting has already been done and that the *Notation* package should not perform any further processing. Typically, this tag is used to surround functions that return expressions which have already been turned into boxes or parsed into expressions.

`TagBox` *tag name*	*visual form*	*effect of tag*
`NotationMadeBoxes` `Tag`	... *subexpression* ...	forces the contents of the surrounded structure to be treated as an expression which is to be processed no further

The visual form and behavior of the tag NotationMadeBoxesTag.

To illustrate, let us consider a simplified version of a notation statement which will be used in §3.4.4 *Definitions for Tensor Formatting*, to generate output in conventional tensor notation. (In the previous subsection we set the notation option `Action` to `PrintNotationRules`, hence we are only examining the generated notation rules.)

```
In[41]:= Notation[Γ_makeGridBox[indices_] ⟸ Tensor[Γ_, indices_?validIndicesQ]]

         NotationMakeBoxes[Tensor[Γ_, indices_?validIndicesQ], StandardForm] :=
           TagBox[RowBox[{Parenthesize[Γ, StandardForm, Times],
                    " ", RowBox[{"makeGridBox", "[",
                          MakeBoxes[indices, StandardForm], "]"}]}], "Tensor"]
```

We see that, as expected, the `NotationMakeBoxes` rule generated by a `Notation[`*lhs* ⟸ *rhs*`]` is of the form.

```
NotationMakeBoxes[rhsExpression] := lhsBoxes                    (3.3.c)
```

The *Notation* package has converted *lhs* to a box structure *lhsBoxes*. Specifically, it converts `makeGridBox[` *indices_* `]` into the following box structure.

```
RowBox[{"makeGridBox", "[",
      Parenthesize[indices, StandardForm, Times], "]"}]
```
(3.3.d)

But this is not what we want. The function `makeGridBox` is designed to take an internal list involving indices together with their high or low positions and convert it to a grid box (to be used to display the upper and lower indices of a tensor). However, in the above rule generated by the *Notation* package, `makeGridBox[` *indices* `]` appears as a literal box structure (3.3.d), not as an expression. Hence, the `makeGridBox` cannot build the requisite grid box of indices. Therefore, we need a way to tell the *Notation* package not to process the expression `makeGridBox[` *indices*`_` `]` (other than to drop the "_") during the construction of the `NotationMakeBoxes` rule. This is what a `NotationMadeBoxesTag` tag box does, as we next see.

```
In[42]:= Notation[Γ_ makeGridBox[indices_] ⟸ Tensor[Γ_, indices_?validIndicesQ]]

      NotationMakeBoxes[Tensor[Γ_, indices_?validIndicesQ], StandardForm] :=
        TagBox[RowBox[{Parenthesize[Γ, StandardForm, Times],
              " ", makeGridBox[indices]}], "Tensor"]
```

The embedded tag box with the tag `NotationMadeBoxesTag` is visually distinguished by a purple background like *subexpression*, due to the tag style `"NotationMadeBoxesWrapperStyle"`. It is evident from the internal definition returned that there is essentially no further processing of the expression `makeGridBox[` *indices*`_` `]` by the *Notation* package. The above `Notation` statement also illustrates the point made earlier in §3.3.1 *Complex Patterns and Notation Pattern Tags*, that *indices*`_?validIndicesQ` is treated as a pattern when it is surrounded by a `NotationPatternTag` tag box.

For our second example, let us consider a simplified version of a `Notation` statement which will be used, in §3.4.5 *Definitions for Tensor Parsing*, to generate the internal form corresponding to a tensor notation input. What it actually does is transform a tensor box structure into an internal tensor expression.

```
In[43]:= Notation[Γ_ stringIndices_?validStringIndicesQ ⟹
           Tensor[Γ_, makeIndices[stringIndices_]]]

      NotationMakeExpression[
           TagBox[RowBox[{Γ_, stringIndices_?validStringIndicesQ}], "Tensor"],
           StandardForm] := MakeExpression[
           RowBox[{"Tensor", "[", RowBox[{Γ, ",", RowBox[{"makeIndices",
                              "[", stringIndices, "]"}]}], "]"}], StandardForm]
```

Observe that, as expected, the `NotationMakeExpression` rule generated by a `Notation[` *lhs* ⟹ *rhs*`]` is of the form

```
NotationMakeExpression[lhsBoxes] := MakeExpression[rhsBoxes]
```
(3.3.e)

Hence, the parts of *rhs* are converted to corresponding parts of the box structure *rhsBoxes*. Specifically, `makeIndices[` *indices*`_` `]` is converted to

```
RowBox[{"makeIndices", "[", indices, "]"}]
```
(3.3.f)

Again, this is not what we want. The function `makeIndices` is designed to take an input grid box of indices and convert that to a row box of indices. Yet, in the above rule the `makeIndices[` *indices_* `]` subpart appears as the box structure (3.3.f), not the expression `makeIndices[` *indices* `]`. Thus, the `makeIndices` performs no evaluation and therefore cannot build the requisite row box of indices. So, as in our first example, we need a way to tell the *Notation* package not to process the expression `makeIndices[` *indices_* `]` (other than to drop the '_') during the construction of the `NotationMakeExpression` rule. And, as in our first example, we can accomplish this by wrapping `makeIndices[` *indices_* `]` with a `NotationMadeBoxesTag` tag box, as we see below.

```
In[44]:= Notation[
           Γ_ indices_?validStringIndicesQ ⟹ Tensor[Γ_, makeIndices[indices_]]]

         NotationMakeExpression[
               TagBox[RowBox[{Γ_, indices_?validStringIndicesQ}], "Tensor"],
               StandardForm] := MakeExpression[RowBox[{"Tensor", "[",
                     RowBox[{Γ, ",", makeIndices[indices]}], "]"}], StandardForm]
```

It is evident from the internal definition returned that there is essentially no further processing of the expression `makeIndices[` *indices* `]` in the generated rules.

So far, we have considered two ways to use a `NotationMadeBoxesTag`. As in the previous subsection with `NotationPatternTag`, there are actually four ways. We could consider either a `Notation[`*lhs* ⟹ *rhs*`]` or a `Notation[`*lhs* ⟸ *rhs*`]` statement; and for each of these, we could surround a part of *lhs* or a part of *rhs* with a `NotationMadeBoxesTag` tag box. Surrounding `f[x_]` and `g[x_]` in the statement below shows that in all four cases, the surrounded expression is essentially left unprocessed.

```
In[45]:= Notation[Γ_ f[x_] ⟺ Tensor[Γ_, g[x_]]]

         NotationMakeExpression[TagBox[RowBox[{Γ_, f[x_]}], Tensor],
               StandardForm] := MakeExpression[RowBox[
                  {"Tensor", "[", RowBox[{Γ, ",", g[x]}], "]"}], StandardForm]

         NotationMakeBoxes[Tensor[Γ_, g[x_]], StandardForm] := TagBox[
               RowBox[{Parenthesize[Γ, StandardForm, Times], f[x]}], Tensor]
```

It is illuminating to compare the behavior of the tag `NotationMadeBoxesTag`, as illustrated above, to that of the tag `NotationPatternTag`, as illustrated next. We do this by using a notation statement which only differs from the above in that it has different tags.

```
In[46]:= Notation[Γ_ f[x_] ⟺ Tensor[Γ_, g[x_]]]

         NotationMakeExpression[TagBox[RowBox[{Γ_, f[x_]}], Tensor],
               StandardForm] := MakeExpression[RowBox[
                  {"Tensor", "[", RowBox[{Γ, ",", g[x]}], "]"}], StandardForm]

         NotationMakeBoxes[Tensor[Γ_, g[x_]], StandardForm] :=
            TagBox[RowBox[{Parenthesize[Γ, StandardForm, Times],
                     Utilities`Notation`Private`makeEvaluatedRowBoxOfBoxes[
                        {f[x]}, StandardForm, None]}], Tensor]
```

Technical Note: Actually, in three out of the four possible cases mentioned, surrounding an expression by a tag box with the tag `NotationPatternTag` will produce the same effect as using a `NotationDoNotProcessTag`. But we have used the two different tags for pedagogical reasons.

Now that we have finished examining the generated rules, let us reset the `Action` option to `CreateNotationRules`.

```
In[47]:= SetOptions[Notation, Action → CreateNotationRules]

Out[47]= {WorkingForm → Automatic, Action → CreateNotationRules}
```

All the information necessary to create complex and sophisticated notations has now been presented. The next section gives a "real world" application of the foregoing material.

3.4 Tensorial Notation

3.4.1 Introduction

As surprising as it may seem, even with the new capabilities provided by the *Notation* package, it is not a trivial exercise to implement an "adequate" notation for tensors. (For references treating tensors, see [81, 240, 295].) This is manifestly evident from the fact that of the many packages dealing with tensors — from those in the fields of general relativity and cosmology (ORTHOCARTAN[199], Classi[7], RSHEEP[298], GRTensorII[248, 249], Redten[144, 145], STENSOR[164, 165]), to general packages in geometry and/or tensor analysis and/or index handling (CARTAN[303, 304], EinS[192, 193], EXCALC[286, 288], MapleTensor[188], MathTensor[258], NP[80], Ricci[202], RicciR[180], Riegeom[266], Riemann[267], SHEEP[115], TTC[15, 56]), and even special packages for index handling in high energy physics (Dill[208], HEPHYS[154]) — none of them have an "adequate" notation for tensors or tensorial objects. Other notational difficulties also arise with these packages. As a result, many physicists feel uncomfortable with the output of these packages, since the notations are unintuitive and different from the notations they are accustomed to using. This obstacle must be surmounted, for in the author's experience, it is important to be able to employ a familiar and intuitive notation when attempting to solve a problem. When researchers unfamiliar with symbolic computation systems attempt to learn a computer algebra system, they will sometimes give up in frustration because they could not figure out how to express their particular problems in the unfamiliar, foreign, and arcane notation of the system. If we can solve the notational problems at the start, we can at least alleviate the language difficulties.

To the above end, and to illustrate some of the more complex underlying principles of the *Notation* package, we will create a working and useful notation for tensors. In actuality, we define a notation for tensorial objects, not just tensors. For us, a *tensorial object* is a tensor or a general indexed object. It need not satisfy any absolute requirement about how its components

transform with respect to coordinate transformations. The transformation properties of specific classes of tensorial objects are prescribed by specific semantics yet to be given.

Just as we did in the section that developed a notation for bras and kets, we must decide what the underlying structure of a *tensor* will be. In §3.4.2 *Prototypical Tensor Box Structure,* we focus on the external representation, hence the underlying box structure, while in §3.4.3 *Prototypical Tensor Expression Structure,* we will deal with the general form of the internal representation. We then proceed to enter the notation in §3.4.4 *Definitions for Tensor Formatting* and §3.4.5 *Definitions for Tensor Parsing.* In the following chapter, we present some simple calculations using these tensors so defined. Real, or industrial strength, applications of our tensorial notation will be deferred until §5 *Prototypical Structures and Quantum Mechanics* and §7 *Tensor Calculus, Applications, and Quasi-Spin,* since we first need to modify the underlying language in §4 *Language Modifications* in order to facilitate the entry of the semantics for the tensorial syntax.

3.4.2 Prototypical Tensor Box Structure

Tensors form a generalized class which include scalars, vectors and matrices as special cases. Scalars, vectors, and matrices have no indices, one index, and two indices, respectively. Tensor representations — from now on we will just call them tensors — have zero or more indices. Normally, in vectors and matrices, all indices are of the same kind. However, with tensors we sometimes need to take into account both *contravariant* (raised) and *covariant* (lowered) indices. (This depends on the metric.) For instance, the following is an example of how a tensor is notationally displayed.

$$\mathbf{T}^{a}_{\ b\,c} \tag{3.4.a}$$

The index a is contravariant and the indices b and c are covariant. **T** is the name of the tensor. The following is a second example of a tensor, with contravariant indices a, b and covariant indices c, d and covariant derivative e.

$$\mathcal{R}^{a\,b}_{\ \ \ c\,d\,;\,e} \tag{3.4.b}$$

To ensure that the tensor name and associated tensor indices will be grouped together in a single structure, we proceed as we did with the bra-ket notation and use a tag box. Also, the natural choice for the indices is a `GridBox`. So as a naive approach, we might try using something like the following.

```
TagBox[RowBox[{tensor name, GridBox[indices]}], "Tensor"]
```
(3.4.c)

Let us give a specific example of this box structure for the tensor `T` with say a raised index `a` and a lowered index `b`, and look at how it would be displayed.

```
In[1]:= TagBox[RowBox[{"T", GridBox[{{"a", ""}, {"", "b"}}]}], "Tensor"] //
        DisplayForm
```

```
Out[1]//DisplayForm=
        a
       T
        b
```

As we can see, the indices are not the right size or in the right place. We can rectify this by changing the font size, shifting the `GridBox`, and using an `AdjustmentBox`. Also, contrary to standard convention, we wish the tensor name to be displayed in bold face, which we accomplish using a `StyleBox`. Our refined candidate would be structured and displayed as follows.

```
In[2]:= TagBox[RowBox[{
                  StyleBox["T", FontWeight -> "Bold"],
                  StyleBox[ AdjustmentBox[
                        GridBox[{{"a", ""}, {"", "b"}}, GridBaseline -> Axis
                           RowSpacings -> 1.4, ColumnSpacings -> 0.3],
                        BoxMargins -> {{-0.35, 0.2}, {0.45, -0.15}},
                        BoxBaselineShift -> -0.45],
                     FontSize -> 9]}], "Tensor"] // DisplayForm
Out[2]//DisplayForm=
        a
       T
        b
```

We can see from the above that we have put the name in bold face, reduced the size of the indices to 9 point, and added the appropriate grid box options. Next, as was the case in the bra-ket notation discussed in §2.7.1 *Prototypical Ket Structures*, we must make sure that the user can select and edit the arguments of a structure — for a tensor these are the name and the indices — yet ensure that the structure itself remains uneditable; and as with the bra-ket notation, we enforce such restrictions using tag boxes. With these modifications, the underlying structure becomes the following.

```
TagBox[RowBox[{
          StyleBox[TagBox[name, "TensorName", Editable -> True,
                Selectable -> True], FontWeight -> "Bold"],
          StyleBox[
             AdjustmentBox[
                TagBox[GridBox[indices,
                      GridBaseline -> Axis, RowSpacings -> 1.4,
                      ColumnSpacings -> 0.3],                          (3.4.d)
                   TensorIndices, Editable -> True,
                   Selectable -> True],
                BoxMargins -> {{-0.35, 0.2}, {0.45, -0.15}},
                BoxBaselineShift -> -0.45],
             FontSize -> 9]}],
    Tensor, Editable -> False,
    Selectable -> False, SyntaxForm -> "symbol"]
```

This structure is sufficient to implement a notation for tensors. However, the box structure is rather complex. As was previously done with the bra-ket example in §2.7.2 *Prototypical Ket Structures using Named Styles*, let us again adopt the highly advantageous practice of specifying the style modifications in a uniform way via the style sheet. Not only does this simplify the underlying structure, but if we wish to change the style or grid adjustments in the way tensors will uniformly appear, we only need perform this modification *once*, in the style sheet, instead of indivdually changing every instance of a tensor in our notebook(s).

To reiterate: adding a new style to the style sheet allows us to set up a uniform style that will be used consistently throughout a notebook. We will define three styles, one for the tensor name, one for the tensor indices, and one for an overall wrapper. The style for the tensor name, `"TensorName"`, will set the font weight and tag box options. The style for the tensor indices, `"TensorIndices"`, will set the font size, adjustment box options, grid box options, and tag box options. Finally, the wrapper style sets the outer tag box options and also the syntax form of the tensor structure as a whole. (These styles were added to this notebook in advance.)

In the style sheet these styles look like

Prototype for style: "TensorName":
TensorName

Prototype for style: "TensorIndices":
TensorIndices

Prototype for style: "TensorWrapper":
TensorWrapper

Examining these cells, by using the **Show Expression** command from the **Format** menu of the front end, we find their structure to be the following.

```
Cell[StyleData["TensorName"],
  Evaluatable->False,
  StyleMenuListing->None,
  FontWeight->"Bold",
  TagBoxOptions->{Editable->True,
  Selectable->True,
  StripWrapperBoxes->True}]
```

```
Cell[StyleData["TensorIndices"],
  Evaluatable->False,
  StyleMenuListing->None,
  FontSize->9,
  AdjustmentBoxOptions->{BoxMargins->{{-0.35, 0.2}, {0.45, -0.15}},
  BoxBaselineShift->-0.45},
  TagBoxOptions->{Editable->True,
  Selectable->True},
  GridBoxOptions->{GridBaseline->Axis,
  RowSpacings->1.4,
  ColumnSpacings->0.3}]
```

```
Cell[StyleData["TensorWrapper"],
  StyleMenuListing->None,
  TagBoxOptions->{Editable->False,
  Selectable->False,
  SyntaxForm->"symbol"}]
```

It is evident that these styles set the correct formatting options.

Recall from §2.6.7 *Style Sheets and Tag Styles*, that once a style, *style name*, has been added to the style sheet, then any box structure that is surrounded by a `StyleBox[..., "style name"]` or a `TagBox[..., tag, TagStyle→"style name"]` will inherit all the options defined in *style name*. For instance, a `GridBox` wrapped by a `StyleBox[ ... , "TensorIndices" ]` will inherit the grid box options `GridBaseline → Axis`, `RowSpacings → 1.4`, `ColumnSpacings → 0.3`.

In light of the foregoing, our prototypical tensor structure should be simplifiable to something like the following.

```
TagBox[RowBox[{
          TagBox[name, "TensorName", TagStyle -> "TensorName"],
          AdjustmentBox[
             TagBox[GridBox[indices], "TensorIndices",
                TagStyle -> "TensorIndices"]]}],
   "Tensor", TagStyle -> "TensorWrapper",
   SyntaxForm -> "symbol"]
```
(3.4.e)

Technical Note: Actually, as previously discussed in §2.7.2 *Prototypical Ket Structures using Named Styles*, it would be truly elegant if adjustment boxes were part of normal styles. In this case, we could omit the adjustment box in the box structure (3.4.e) and instead incorporate it into the `TensorIndices` style. Also, we must again explicitly include the syntax form option in the outer tag box to circumvent the parenthesizing bug in *Mathematica* as previously described in §2.7.2 *Prototypical Ket Structures using Named Styles*.

Unfortunately, due to current limitations in *Mathematica*, there are some problems with the above box structure (3.4.e). Firstly, recall that adjustment boxes with no adjustments are automatically stripped out, hence we need to add at least one adjustment, like `BoxBaseline-Shift → -0.45`. Secondly, for some mystifying reason, including a style box around the adjustment box or some similar code seems to be necessary in order that the front end behave correctly when moving a cursor through the tensor box (this will hopefully be rectified with the next release). Therefore, our prototypical tensor will have the following structure.

```
TagBox[RowBox[{
          TagBox[name, "TensorName", TagStyle -> "TensorName"],
          StyleBox[AdjustmentBox[
                TagBox[GridBox[indices], "TensorIndices",
                   TagStyle -> "TensorIndices"],
                BoxBaselineShift -> -0.45], "TensorIndices"]}],
   "Tensor", TagStyle -> "TensorWrapper",
   SyntaxForm -> "symbol"]
```
(3.4.f)

One might think that using uniform styles is not worth the extra complication of having to modify the style sheet. However, it cannot be stressed enough how important this is for long term viability and program management. It is extremely worthwhile for a variety of reasons.

For example, when printing, one might want smaller indices than for on-screen editing, or a specific user might not want the tensor name to be bold, etc. Using defined styles allows greater flexibility and enforces a cleaner structure. It is *good* style.

3.4.3 Prototypical Tensor Expression Structure

After having chosen the underlying box structure for tensors, the next step is to determine how the full form tensor expressions should be structured. There are two obvious candidates that are immediately apparent to mind. The first candidate representation we will consider for a tensor expression structure is:

$$\texttt{Tensor[}\textit{name}\texttt{, \{}\textit{index}_1\texttt{, }\textit{index}_2\texttt{, ..., }\textit{index}_i\texttt{\},}\\ \texttt{\{}\textit{elevation}_1\texttt{, }\textit{elevation}_2\texttt{, ..., }\textit{elevation}_i\texttt{\}]} \tag{3.4.g}$$

For instance, the tensor $\mathbf{T}^{a}_{\ bc}$ would be represented as `Tensor[ T, {a, b, c}, {High, Low, Low} ]`. There are some benefits to such an expression structure, and indeed it would be adequate. However, creating a notation that uses the above underlying expression structure will be left as an exercise for the reader. (It is suggested that one finish reading this section before attempting such a notation.)

The second expression structure we consider is:

$$\texttt{Tensor[}\textit{name}\texttt{, \{}\textit{elevation}_1\texttt{[}\textit{index}_1\texttt{], }\textit{elevation}_2\texttt{[}\textit{index}_2\texttt{], ..., }\textit{elevation}_i\texttt{[}\textit{index}_i\texttt{]\}]} \tag{3.4.h}$$

For instance, the tensor $\mathbf{T}^{a}_{\ bc}$ would be represented as `Tensor[ T, {High[a], Low[b], Low[c]} ]`. It is this second expression structure which we will use to represent our tensors, since it is easy to work with when pattern matching. Incidentally, this structure is close to the data structure the package Ricci[202] uses. There are obvious variants of our chosen expression structure, such as `Tensor[ T, {{a,High}, {b,Low}, {c,Low}]} ]` or even `Tensor[ T, {a[High], b[Low], c[Low]} ]`. However, since these variants can be treated similarly, we will not discuss them further. Our chosen representation is consequently the following.

`Tensor[`*name*`,` *indices*`]`	represents a tensor called *name* with the list of indices *indices*

The structure of a tensor.

In the spirit of the *Notation* package, instead of using `High[` *index*`]` or `Low[` *index*`]` in our definitions, let us introduce the following notations for indices that are contravariant or covariant.

```
In[3]:= Notation[a_^+ ⟺ High[a_]]
        Notation[a_^- ⟺ Low[a_]]
```

For our indices, we can summarize the data structures for them, the notations for them, and their meanings in the following table.

data structure	*notation*	*purpose*
`High[`*index*`]`	*index*$^{+}$	represents a contravariant index in a *tensor*
`Low[`*index*`]`	*index*$^{-}$	represents a covariant index in a *tensor*

The data structures and notations for indices.

Now that we have chosen an underlying box structure and an underlying expression structure for representing tensors, a problem arises. In the case of the bra-ket notation, the box structure and the expression structure were simple enough and compatible enough that defining a notation for bras, kets, and bra-kets was relatively easy. Unfortunately, the same is not true for tensors. The incompatibility between the tensor box structure and expression structure lies in the indices. To illustrate this, let us examine the indices in $\mathbf{T}^{a}_{\ bc}$ for both structures. As a part of the tensor box structure, the indices are represented by

```
GridBox[{{"a", "", ""}, {"", "b", "c"}}]
```
(3.4.i)

But as part of the tensor expression structure, the indices are represented by

```
{High[a], Low[b], Low[c]}
```
(3.4.j)

It is clear that there is no trivial conversion between the box structure and the expression structure. Herein lies the problem. The complications that arise from the conversion from boxes to expressions and vice-versa occupy the next two subsections.

3.4.4 Definitions for Tensor Formatting

In the two previous subsections, we decided on both an underlying box structure and an underlying expression structure for tensors. We will now proceed, in a top down fashion, to implement a *Mathematica* notation for tensors, using two `Notation` statements connecting the box structures and expression structures representing tensors. The implementation of this notation is non-trivial because there is no direct translation between the two structures.

Let us implement the notation that transforms a tensor expression, such as `Tensor[Γ, {a⁺, b⁻, c⁻}]`, into a tensor box structure, and then discuss it.

```
In[5]:= Notation[Γ_ makeGridBox[indices_] ⟸ Tensor[Γ_, indices_?validIndicesQ]]
```

In this `Notation` statement, *indices*`_?validIndicesQ` is wrapped by a tag box with the tag `NotationPatternTag`, and the expression `makeGridBox[` *indices*`_ ]` is wrapped by a `NotationMadeBoxesTag` tag box. This latter wrapping is necessary since `makeGridBox` returns a box structure, hence we must not allow `MakeBoxes` to process it further. (Recall from §3.3.5 *NotationMadeBoxesTag*, that if a substructure is wrapped with a tag box with the tag `NotationMadeBoxesTag`, then in the rules generated by the *Notation* package, the substructure will not be surrounded by `MakeBoxes`.)

To confirm that the NotationMadeBoxesTag has functioned correctly, let us inspect the NotationMakeBoxes rule generated by the above Notation statement.

```
In[6]:= Notation[Γ_ makeGridBox[indices_] ⟸ Tensor[Γ_, indices_?validIndicesQ],
          Action → PrintNotationRules]

        NotationMakeBoxes[Tensor[Γ_, indices_?validIndicesQ], StandardForm] :=
          TagBox[RowBox[{TagBox[MakeBoxes[Γ, StandardForm],
                        "TensorName", TagStyle → "TensorName"],
                    StyleBox[AdjustmentBox[TagBox[makeGridBox[indices],
                              "TensorIndices", TagStyle → "TensorIndices"],
                            BoxBaselineShift → -0.45], "TensorIndices"]}],
            "Tensor", TagStyle → "TensorWrapper",
            SyntaxForm → "symbol"]
```

It is evident that the NotationMadeBoxesTag functioned correctly, since makeGridBox[*indices_*] does not have a MakeBoxes wrapper surrounding it, unlike say the Γ. It is also clear that we will need to define a testing function, validIndicesQ, that will determine if the indices in the tensor expression are of the right form to be translated into a grid box of indices. In addition, it is necessary to define the function makeGridBox, which will create a box structure of indices from a given list of indices.

First we consider validIndicesQ. This function determines whether each of the indices in the tensor expression is of the form Up[_] or Down[_]. Obviously, if one of the indices is not of this form, then we cannot translate from a tensor expression to a tensor box structure.

```
In[7]:= validIndicesQ @ {indices___?(MatchQ[#, High[_] | Low[_]] &)} := True;
        validIndicesQ @ _ := False;
```

As a quick test of validIndicesQ we have

```
In[9]:= validIndicesQ /@ {{a⁺, b⁻, c⁻}, {a, b}, {a, b⁻}}

Out[9]= {True, False, False}
```

Once it has been determined, via validIndicesQ, that the indices are of the correct form, it is the duty of makeGridBox to create a grid box of indices from the given list of indices.

```
In[10]:= makeGridBox @ indices_List :=
           GridBox @ Transpose [makeStringIndexPair /@ indices];

In[11]:= makeStringIndexPair @ High @ index_ := {MakeBoxes @ index, ""};
         makeStringIndexPair @ Low @ index_ := {"", MakeBoxes @ index};
```

Technical Note: Although it is not strictly necessary for the present work, the full source code for the *Tensors`* package sets the HoldAll attribute for the functions validIndicesQ, makeGridBox, and makeStringIndexPair; and also suitable ensures that no unwanted evaluation takes place.

With the notation declaration above and the subsidiary functions validIndicesQ and makeGridBox, our tensors are now being formatted properly, as demonstrated by the following.

```
In[13]:= Tensor[Γ, {a⁺, b⁻, c⁻}]
```

```
Out[13]= Γ^a
           b c
```

```
In[14]:= Tensor[Γ, {a⁺}]
```

```
Out[14]= Γ^a
```

Actually, it is commonplace in physics to denote the partial derivative by a comma ','. (See any of the references on relativity such as [81, 240, 255] or quantum field theory [138, 173, 186, 278] or mechanics [130].) Also, in general relativity and differential geometry one uses covariant derivatives, denoted by ';' — see [81, 240, 255]. Let us add these to the parsing of tensors.

```
In[15]:= makeStringIndexPair @ Low[";"] := {"", ";"};
         makeStringIndexPair @ Low[","] := {"", ","};
```

```
In[17]:= Tensor[Γ, {a⁺, b⁻, c⁻, ";"⁻, b⁻}]
```

```
Out[17]= Γ^a
           b c ; b
```

For completion, observe that products of tensors are formatted correctly, and in particular, are parenthesized correctly since the syntax form of the overall box structure of each tensor structure is a symbol — cf. (3.4.f).

```
In[18]:= Tensor[ϵ, {a⁻, d⁻, e⁻}] Tensor[Γ, {a⁺, b⁻, c⁻}]
```

```
Out[18]= Γ^a
           b c  ϵ
                  a d e
```

Formatting is now working, so we next turn to parsing.

3.4.5 Definitions for Tensor Parsing

The reciprocal notation that transforms a tensor box structure into a tensor expression is slightly more complicated than that for the formatting of tensor expressions. In the notation statement below, both *stringIndices_*`?validStringIndicesQ` and `makeIndices[` *stringIndices_* `]` are each wrapped by a tag box with the tag `NotationPatternTag`. This is to ensure, as discussed in §3.3.1 *Complex Patterns and Notation Pattern Tags*, that they are interpreted as patterns.

```
In[19]:= Notation[
           Γ_ stringIndices_?validStringIndicesQ ⟹ Tensor[Γ_, makeIndices[stringIndices_]]]
```

In essence, we only want to match boxes that can be interpreted as a tensor. Let us inspect the internal rules generated by the above statement.

```
In[20]:= Notation[
           Γ_ stringIndices_?validStringIndicesQ ⟹ Tensor[Γ_, makeIndices[stringIndices_]],
           Action -> PrintNotationRules]
```

```
NotationMakeExpression[TagBox[
        RowBox[{TagBox[Γ_, "TensorName", TagStyle → "TensorName"],
              StyleBox[AdjustmentBox[
                    TagBox[stringIndices_?validStringIndicesQ,
                       "TensorIndices", TagStyle → "TensorIndices"],
                    BoxBaselineShift → -0.45], "TensorIndices"]}],
        "Tensor", TagStyle → "TensorWrapper",
        SyntaxForm → "symbol"],
     StandardForm] :=
  MakeExpression[
     RowBox[
        {"Tensor", "[", RowBox[{Γ, ",", makeIndices[stringIndices]}],
           "]"}], StandardForm]
```

We will proceed in a top down fashion. From the above, it is clear we will have to define a testing function, `validStringIndicesQ`, which will determine if a grid box of potential indices can actually be interpreted as a valid bunch of tensor indices. In more detail, `validStringIndicesQ` checks to see that the grid box contains a valid list of index pairs. It ignores style boxes. We must also define the function `makeIndices`, which will create the box structure of the parsed indices from the grid box of indices.

```
In[21]:= validStringIndicesQ @ GridBox[ indices_List, ___] :=
           And @@ (validStringIndexPairQ /@ Transpose @ indices)
         validStringIndicesQ @ StyleBox[ boxes_, ___] :=
           validStringIndicesQ @ boxes
         validStringIndicesQ @ other_ := False
```

A valid index pair is something like {*index, white space*} or {*white space, index*}, that is, something that can be interpreted as a high or low index.

```
In[24]:= validStringIndexPairQ @ {_?whiteSpaceQ, _?validStringIndexQ} := True
         validStringIndexPairQ @ {_?validStringIndexQ, _?whiteSpaceQ} := True
         validStringIndexPairQ @ _ := False
```

A valid index is something that is parsable or a ',' or a ';' — where, as is conventional in tensor analysis, the latter two symbols denote partial and covariant differentiation, respectively — see [81, 240, 255, 326, 333].

```
In[27]:= validStringIndexQ[";" | "," | a_?silentParsableQ] := True
         validStringIndexQ @ other_ := False
```

Apart from a definition of `whiteSpaceQ` and `silentParsableQ`, which are given shortly, we have provided a complete definition for `validStringIndicesQ`.

Once it has been determined, via `validStringIndexQ`, that a grid box of potential indices actually represents tensor indices, it falls to `makeIndices` to transform the grid box of indices into a row box of `High` and/or `Low` indices. (Style boxes are ignored by `makeIndices`.)

```
In[29]:= makeIndices @ StyleBox[ boxes_, ___] := makeIndices @ boxes
         makeIndices @ GridBox[ indices_List, ___] :=
           RowBox @ {"{", RowBox @ padList[
                        parseStringIndices /@ Transpose @ indices, ","], "}"}
         makeIndices @ other_ := ErrorBox @ other
```

The purpose of `parseStringIndices` is to convert each column of the grid box into a single `High` or `Low` type index.

```
In[32]:= parseStringIndices @ {_?whiteSpaceQ, index_?validStringIndexQ} :=
           RowBox @ {"Low", "[", parseSpecial@ index, "]"}
         parseStringIndices @ {index_?validStringIndexQ, _?whiteSpaceQ} :=
           RowBox[{"High", "[", parseSpecial@index, "]"}]
         parseStringIndices @ other_ :=
           RowBox @ {"tensorParseError", "[", other, "]"}
```

The function `parseSpecial` ensures that the special characters for covariant and partial differentiation, as well as ordinary indices, are handled correctly.

```
In[35]:= parseSpecial @ ";" := "\";\""
         parseSpecial @ "," := "\",\""
         parseSpecial @ other_ := First @ StripBoxes @ other
```

We have now defined a notation for parsing tensors, that is, a notation that translates tensor box structures into tensor expressions. It only remains to define `padList`, `whiteSpaceQ` and `silentParsableQ`.

The function `padList` will insert a padding element, *padElement*, between all adjacent members of a list. This is useful for inserting, say, commas or infix operators, into a box structure.

```
In[38]:= padList[list_List, padElement_] :=
           Drop[#, -1] & @ Flatten @ Thread @ {list, padElement}
```

Here is an example using `padList`.

```
In[39]:= padList[{"a", "b", "c"}, ","]

Out[39]= {"a", ",", "b", ",", "c"}
```

For the upcoming function definitions, it is convenient to reintroduce the following infix notations for `SameQ` and `UnsameQ`.

```
In[40]:= InfixNotation[≠, UnsameQ]
         InfixNotation[≡, SameQ]
```

To determine if a given string consists of only white space characters, that is things like tabs, spaces, empty strings, etc., we require the function `whiteSpaceQ`.

```
In[42]:= whiteSpaceQ @ string_String :=
           DeleteCases[Characters @ string,
            ("\t" | "\n" | " " |
                      "" | "" | "" | "" | "" | "" | "" |
                     "" | "" | "\n" | "" | " " | "⁚" | "␣" | "␣" | "" | "")] ≡
             {};
         whiteSpaceQ @ other___ := False
```

The last function, `silentParsableQ`, is a variant of `parsableQ`. The function `parsableQ` determines whether a given box structure or string is parsable in `StandardForm`. If it is not parsable, then the error message given by the first non-parsable object will be displayed and `parsableQ` will return `False`.

```
In[44]:= parsableQ @ boxexpr_ :=
           Head @ ToExpression[boxexpr, StandardForm, HoldComplete] ≡
             HoldComplete
```

For example, the box structure of a_+ is parsable while that of $a^{\int}$ is not

```
In[45]:= parsableQ[SubscriptBox["a", "+"]]

Out[45]= True

In[46]:= parsableQ[SuperscriptBox["a", "∫"]]

         Syntax::sntxi : Incomplete expression; more input is needed.

         ToExpression::esntx :
           Could not parse SuperscriptBox[a, ∫] as Mathematica input.

Out[46]= False
```

The function `silentParsableQ` performs exactly the same function as `parsableQ`, but it will not report *any* error messages. We accomplish this using `silentEvaluate` which, like `Evaluate`, will evaluate an expression and return the result, but unlike `Evaluate`, it will not report *any* error messages.

```
In[47]:= SetAttributes[silentEvaluate, HoldAll];
         silentEvaluate @ expr_ :=
           Block[{Message}, SetAttributes[Message, HoldFirst]; expr];
         silentParsableQ @ boxexpr___ := silentEvaluate @ parsableQ @ boxexpr;

In[50]:= silentParsableQ[SubscriptBox["a", "+"]]

Out[50]= True

In[51]:= silentParsableQ[SubscriptBox["a", "∫"]]

Out[51]= False
```

Finally, let us declare that any tensor object that is not parsable by the above is illegal.

```
In[52]:= MakeExpression[TagBox[any_, Tensor, opts___], StandardForm] := $Failed;
```

Tensor parsing now works!

```
In[53]:= T^a_bc // FullForm

Out[53]//FullForm=
         Tensor[T, List[High[a], Low[b], Low[c]]]
```

3.4.6 Concluding Remarks

To conclude this section, observe that, as desired, our tensor notation is working in both input and output.

```
In[54]:= T^a_bc

Out[54]= T^a_bc

In[55]:= FullForm @ %
Out[55]//FullForm=
Tensor[T, List[High[a], Low[b], Low[c]]]

In[56]:= Tensor[ϵ, {a⁻, d⁻, e⁻}] Tensor[Γ, {a⁺, b⁻, c⁻}]

Out[56]= Γ^a_bc ϵ_ade
```

We are now in the position where we have created a notation for tensors. Some rather complex concepts were needed in the two previous sections, §3.2 *Principles behind the Notation Package* and §3.3 *Complex Patterns in Notations*, as well as in the previous chapter, §2 *The Notation Package*. But even though the end result may seem relatively straightforward, this accomplishment should not be underrated. Currently, to the author's knowledge, there are no other competing systems that have such facilities. This is even despite the fact that *Mathematica* 3.0 was released some 3 years ago, in 1996. Moreover, the lengthy history of the field, which started in the 1960s — see d'Inverno[82] — and the fact that many programs have tackled such problems are testament to the fact that the problem is non-trivial.

No less than six *Mathematica* packages for tensor analysis have been introduced or updated since *Mathematica* 3.0 was released: MathTensor[258], Ricci[202], Cartan[303, 304], TTC[15, 56], EinS[192], and GRTensorII[248, 249]. There are also other associated *Mathematica* packages that could make good use of the "correct" display of general indicial expressions: among these are Dill[208], FeynCalc[230, 231], HEPHYS[154], HIP[168], and Tracer[174]. This listing of programs, of course, does not include any of the programs or systems for handling tensors not based in *Mathematica*, such as STensor[165], Redten[144, 145], Riegeom[266], or EXCALC[286, 288]

None of these aforementioned systems properly handle the notations for tensors. Ours does. Obviously, we have achieved something highly desirable.

In the next section, we turn our attention to some simple issues involving, as well as examples demonstrating, the use of our notation for tensorial objects.

3.5 Tensors: Examples and Ancillary Notations

3.5.1 Introduction

At this juncture, we are faced with a choice. Should we present some actual calculations before we have modified our underlying language and hence give somewhat awkward code? Or should we delay any calculations until after §4 *Language Modifications*, in which case we will be able to perform them "properly"? There are advantages and disadvantages to both approaches. Nevertheless, it is perhaps beneficial to see how to do calculations the "wrong way" in order to recognize and appreciate the advancements made when we perform and structure calculations the "right way".

To the above end, in this section we will consider examples that use the tensor notation and, more generally, use the notation for indexed objects which need not represent tensors. We will also develop some new ancillary tensor notations. Specifically, in §3.5.2 *Simple Examples Using Tensors*, we implement and illustrate a small precursor to dummy index handling that we need for tensor analysis. In §3.5.3 *Dummy Indices*, we continue our treatment of dummy indices by introducing a function for reindexing, along with some notations for "tensorial assignment". Next, in §3.5.4 *Complications Involving Index Elevations*, we consider an example from quantum mechanics involving indexed objects with negative indices. We proceed on to a simplistic system for Cartesian vector calculus in §3.5.5 and §3.5.6. This motivates the notational extension we give in §3.5.7 *Extended Tensor Syntax*. Finally, in §3.5.8 *Examples Using the Extended Tensor Syntax*, we conclude this section with an example demonstrating the use of our extended syntax: an implementation of the generalized Kronecker delta function.

To reiterate, the examples in this section are really for illustration of the tensorial notation. They are not intended to provide a complete solution to the problems illustrated.

3.5.2 Simple Examples Using Tensors

Let us "play with" a few simple calculations in this subsection, demonstrating the tensorial notation. Later, in §7 *Tensor Calculus, Applications, and Quasi-Spin*, we will give a "proper" functioning to go with the tensor notation.

Here is a simple tensorial expression.

```
In[1]:= Tensor[ϵ, {a⁻, d⁻, e⁻}] Tensor[Γ, {a⁺, b⁻, c⁻}]
```

```
Out[1]= Γ^a_bc ϵ_ade
```

We can add two tensors and, in general, perform computations on expressions involving tensors.

```
In[2]:= % + Γ^a_bc ϵ_dea

Out[2]= Γ^a_bc ϵ_ade + Γ^a_bc ϵ_dea

In[3]:= Simplify @ %

Out[3]= Γ^a_bc (ϵ_ade + ϵ_dea)
```

When dealing with tensors in relativity, mechanics, quantum mechanics, and indeed almost any discipline, the Einstein summation convention is often used. That is, when two indices appear in a product, it is assumed that these indices are summed over. (This material should be standard to any physicist — see, for instance, Simmonds [295] or D'Inverno[81].) However, in the context of symbolic computation, a problem arises, called *the reindexing problem*. To illustrate this problem, consider the expansion of the Christoffel symbol in terms of the metric when the metric is torsion free[81, 240, 295].

$$\Gamma^{i}_{\ kl} = \frac{1}{2} g^{im} \left(g_{km,l} + g_{ml,k} - g_{kl,m} \right) \tag{3.5.a}$$

Now, if we are asked to expand, say, the Christoffel symbol $\Gamma^{l}_{\ mn}$, how do we proceed? We cannot blithely use formula (3.5.a), since then we would have the following reduction.

$$\Gamma^{l}_{\ mn} \longrightarrow \frac{1}{2} g^{lm} \left(g_{mm,n} + g_{mn,m} - g_{mn,m} \right) \tag{3.5.b}$$

Obviously this is nonsense, since the index m appears twice in a covariant position in the same metric tensor. Thus, as physicists, we automatically rename variables, almost unconsciously, whenever using formulas in order to maintain the Einstein summation convention. That is, we would pick an unused index, say a, and write the following.

$$\Gamma^{l}_{\ mn} \longrightarrow \frac{1}{2} g^{la} \left(g_{ma,n} + g_{an,m} - g_{mn,a} \right) \tag{3.5.c}$$

A solution to this problem in the context of symbolic computation is needed. Evidently, we must somehow encompass automatic renaming or reindexing whenever we rewrite expressions.

Far better ways to resolve index handling will be given later, but for now let us proceed in an extremely simplistic fashion. We will directly create a rule that replaces any summed indices on the right hand side with unique indices, that is, indices never used before in the session. We do this using the `Module` construct.

```
In[4]:= expandΓ = {Γ^i_ _k_ _l_ :> Module[{m}, 1/2 g^im (g_km,l + g_ml,k - g_kl,m)]};
```

Consequently, calculations using expandΓ can now be tackled. For example, here is the expansion of the covariant derivative $T_{a\,;b}$.

```
In[5]:= T[a, b] + T[c] Γ[c, a, b]

Out[5]= T[a, b] + T[c] Γ[c, a, b]
```

We expand the Christoffel symbol in this expression as follows.

```
In[6]:= % /. expandΓ // Expand

Out[6]= -(1/2) g[c, m$1016] g[a, b, m$1016] T[c] +
          (1/2) g[c, m$1016] g[a, m$1016, b] T[c] + (1/2) g[c, m$1016] g[m$1016, b, a] T[c] + T[a, b]
```

It is startlingly obvious that we are going to need a function to reindex our expressions after we have chosen unique indices. The obvious choice is to replace the "ugly" dummy index above by a "nice" index. Using "replacements" was the way calculations typically proceeded before the material in this thesis. Although it is possible to make do with this style of calculation, it is certainly not elegant. For instance, in EinS[192] the expansion of Christoffel symbols in terms of the metric and its derivatives would be accomplished by the following rule — which is at best arcane, and at worst ugly and inelegant.

```
withmetric =
   {Gam[i_, k_, l_] :→
         Module[{m},
            DefES[1 / 2 gc[i, m] * (PD[g[m, k], x[l]] +
                        PD[g[m, l], x[k]] - PD[g[k, l], x[m]]),
               {m}, ESRange → $ESDimension]]};
```
(3.5.d)

Observe the bad handling of the elevation of the indices, the direct specification of the module in order to scope the variables (as above) and other problems which are apparent to the experienced practitioner. Unlike (3.5.d), at least our version, given by the rule expandΓ, is readable. But we can easily do much better, as will be seen in the next subsection. Later still, in §7 *Tensor Calculus, Applications, and Quasi-Spin*, we shall tackle and overcome the other intrinsic problems with the approach that uses "replacements".

3.5.3 Dummy Indices

The previous subsection presented several simple examples using tensors/indexed objects. It is obvious that we need further tools to manipulate such objects. Most tensor systems share at least some tools in common. Almost invariable, one of these tools is that for *reindexing* a tensorial expression. In practice, this amounts to replacing, in a uniform way, all the dummy indices by new dummy indices. In §6 *An Algorithm for Tensor Simplification*, we will extensively revisit tensors, their functioning, and present an algorithm for canonicalizing tensor expressions, all in the context of developing what will be called the *Tensors* package. For now though, let us just use the *Tensors* package and its ancillary routines. The following clears all the variables we have so far introduced and then loads the *Tensors* package as well as the common notations.

```
In[7]:= ClearNotations[];
        ClearAll /@ Names @ "Global`*";
        Remove /@ Names @ "Global`*";

In[10]:= << Tensors`
         << CommonNotations`
```

Loading the package defines many top level functions. The number of such functions has been kept to a minimum in order to keep the interface clean and functional. As already mentioned, we will discuss these functions in further detail later, but for now let us just use them with cavalier flippancy.

In order to handle indexed objects, we must first introduce at least one set of indices. Loading the package automatically accomplishes this by declaring that the following symbols are *general indices*: `a`, `b`, `c`, `d`, `i`, `j`, `k`, `l`, `m`, `n`, `o`, `p`, `q`. We can, of course, alter this set or indeed work with different sets, for example α, β, γ, etc. However, these simple details are left until §7.2.1 *Indices And Coordinates*, when we will have dispensed with all our notational concerns. Under most circumstances, it is not critical which class of indices one uses for which function. In fact, to demonstrate this in the early sections of this thesis, we will use general indices. In later chapters, we will use more specialized indices belonging to other classes.

Let us return to *reindexing* expressions, for which we use the *Tensors* package function `ReIndex`. For instance, if we reindex the equivalent expressions $\mathbf{T}_{i}\,\mathbf{\Gamma}^{i}{}_{ab}$ and $\mathbf{T}_{d}\,\mathbf{\Gamma}^{d}{}_{ab}$, then we should obtain the same expression, with the dummy indices `i` and `d` being replaced by a common dummy index. `ReIndex` accomplishes this, using the first general index which is not free in either expression, namely `c`.

```
In[12]:= ReIndex[T_i Γ^i_ab + T_d Γ^d_ab]

Out[12]= 2 T_c Γ^c_ab
```

In fact, reindexing can be thought of as a partial step towards canonicalization. That is, some expressions which initially appear different, but which are in actuality equivalent, can be

transformed into the same expression just by renaming the dummy indices in an expression in a uniform way.

Recall the example from the previous subsection, where we expanded the Christoffel symbol in the covariant derivative of $T_{a\,;b}$.

```
In[13]:= expandΓ = {Γ^i_ _k_ _l_ :> Module[{m}, 1/2 g^i m (g_k m , l + g_m l , k - g_k l , m)]};

In[14]:= T_a , b + T_c Γ^c _a b /. expandΓ // Expand

Out[14]= -1/2 g^c m$1365 g_a b , m$1365 T_c +
          1/2 g^c m$1365 g_a m$1365 , b T_c + 1/2 g^c m$1365 g_m$1365 b , a T_c + T_a , b
```

This expression can now be reindexed to obtain a normal looking tensor. As above, `ReIndex` uses the first general index, this time d, which is not free in the relevant expression.

```
In[15]:= ReIndex @ %

Out[15]= -1/2 g^c d g_a b , d T_c + 1/2 g^c d g_a d , b T_c + 1/2 g^c d g_d b , a T_c + T_a , b
```

Unfortunately, reindexing expressions is an intrinsic property of tensor manipulation that cannot readily be avoided. This issue will be revisited in §7 *Tensor Calculus, Applications, and Quasi-Spin*. In that chapter, we will outline the reasons and motivations which lead to the following notation and definition for *tensorial assignment*.

```
In[16]:= Notation[lhs_ :≔ rhs_ ⟺ TensorSetDelayed[lhs_, rhs_]];

In[17]:= (lhs_ :≔ rhs_) :=
           With[{dummies = DummyIndices[rhs] \ DummyIndices[lhs]},
             lhs := Module[dummies, rhs]]
```

For now though, a tensor assignment, such as *lhs* :≔ *rhs*, just makes a delayed assignment of *rhs* for *lhs*, with the proviso that the dummy indices in the rhs will be *unique in every usage instance*. For example only, here is an extremely simplistic toy definition of the covariant derivative which works only for vectors (and is in no way intended to be part of a more general solution).

```
In[18]:= T_ i_ ; j_ :≔ T_ i , j + Γ^k _i j T_k
```

We can easily illustrate the uniqueness of the dummy index used to replace k in each application of the rule $T\_\_{i\_\,;j\_} \to T_{i,j} + \Gamma^{k}_{\;ij}\,T_k$.

```
In[19]:= U_a ; b + U_a ; b

Out[19]= 2 U_a , b + U_k$1381 Γ^k$1381 _a b + U_k$1382 Γ^k$1382 _a b
```

Of course, these individual terms with unique indices reindex to the same object.

```
In[20]:= ReIndex @ %
```

$$\text{Out[20]=}\quad 2\,\mathbf{U}_{a,b} + 2\,\mathbf{U}_{c}\,\mathbf{\Gamma}^{c}{}_{ab}$$

Using tensorial assignment, we can elegantly define the Christoffel symbol in terms of the metric tensor.

$$\text{In[21]:=}\quad \mathbf{\Gamma}^{i\_}{}_{k\_\,l\_} := \frac{1}{2}\,\mathbf{g}^{i\,m}\left(\mathbf{g}_{k\,m,\,l} + \mathbf{g}_{m\,l,\,k} - \mathbf{g}_{k\,l,\,m}\right)$$

Observe that this definition is marvelously elegant in its simplicity. We are finally starting to enter formulas that "look" and "act" like a physicist would expect. Whenever expressions using the Christoffel symbol are used, the dummy indices of the expanded form will always be unique.

$$\text{In[22]:=}\quad \mathbf{\Gamma}^{i}{}_{j\,m}\,\mathbf{\Gamma}^{m}{}_{i\,k}$$

$$\text{Out[22]=}\quad \frac{1}{4}\,\mathbf{g}^{i\,m\$1383}\,\mathbf{g}^{m\,m\$1384}\left(-\mathbf{g}_{j\,m,\,m\$1383} + \mathbf{g}_{j\,m\$1383,\,m} + \mathbf{g}_{m\$1383\,m,\,j}\right)\left(-\mathbf{g}_{i\,k,\,m\$1384} + \mathbf{g}_{i\,m\$1384,\,k} + \mathbf{g}_{m\$1384\,k,\,i}\right)$$

We can, of course, reindex this expression to obtain a "normal" looking tensor.

```
In[23]:= ReIndex @ %
```

$$\text{Out[23]=}\quad \frac{1}{4}\,\mathbf{g}^{a\,c}\,\mathbf{g}^{b\,d}\left(\mathbf{g}_{a\,d,\,k} - \mathbf{g}_{a\,k,\,d} + \mathbf{g}_{d\,k,\,a}\right)\left(\mathbf{g}_{c\,b,\,j} - \mathbf{g}_{j\,b,\,c} + \mathbf{g}_{j\,c,\,b}\right)$$

It should be evident that the *Notation* package allows us to change many features of our underlying language, or at least give the appearance of such changes. The expressive power we can harness from a good notation should not be underestimated. Indeed, this raises a philosophical question. Does the language remain "unchanged"? Or do the semantics we give our notations imply that the grammar extensions as a whole are more than just syntactic sugar? As is the case in philosophy, there is no correct answer — just viewpoints.

3.5.4 Complications Involving Index Elevations

Some packages attempt to do some of the same things as our *Tensors* package. Indeed, some packages in certain areas have capabilities that are as yet unimplemented in the package under development here. However, in criticism, these other packages are, at times, "clunky". Their lack of notation gives rise to problems — for which they should not be criticized since such features were at the time of development, and in many cases still are, beyond their host systems' software. In this section, let us focus on just one of these problems: how to denote the elevation of indices.

In an attempt to capture the covariant versus contravariant nature of an index, many of the packages denote a covariant index by −index and a contravariant index by +index. For these, `T[a, -b, -a, c]` would represent the tensor $T^{a\ \ c}_{\ b\ a}$. Amongst such systems which follow this convention are EXCALC[286, 288], DUMMY[153] and its successor CANTENS[52], Riegeom[266], and TTC[15, 56]. Of course, such a convention leads to disaster if the indices that are being summed over are not all intrinsically positive. Indeed, this situation frequently arises outside general relativity. (In all fairness, it appears that many of the systems mentioned were not designed to have applicability outside relativity and/or differential geometry.) For instance, in treating angular momentum in quantum mechanics, we often use indices which contain angular momentum quantum numbers that can range from $-J$ to J. A typical example arises when we actively transform a set of angular momentum eigenkets by a rotation. This rotation results in transforming each eigenket into a linear combination of the original eigenkets, with the coefficients of the linear combination providing the indexed objects. (General details of finite rotations of quantum systems are given in most standard texts on Quantum Mechanics. For example, see Shankar[292], Merzbacher[232], or Cohen-Tannoudji[68]; or more specifically, see Thompson[311], Weissbluth[334], Judd[179], or Fano[106].)

```
In[24]:= Notation[D^(j_)[ω__] ⟺ FiniteRotation[j_, ω__]]
```

Let us load the bra-ket notation created in §2.7 *Example: Bra-Ket Notation*.

```
In[25]:= << Notations`Braket`
```

We can now define how rotations act on our kets.

```
In[26]:= P[ω_] · |γ___, j_ĵ, m_Ĵz⟩ := Σ_{n=-j}^{j} D^(J)[ω]_{n m} |γ, j_ĵ, m_Ĵz⟩
```

This is a perfectly valid formula which uses "negative indices". Here is a typical expansion.

```
In[27]:= P[ω] · |2_ĵ, 1_Ĵz⟩
```

$$\text{Out[27]}= |2_{\hat{\jmath}},\ 1_{\hat{J}_z}\rangle\, \mathbf{D}^{(2)}[\boldsymbol{\omega}]_{-2\ 1} + |2_{\hat{\jmath}},\ 1_{\hat{J}_z}\rangle\, \mathbf{D}^{(2)}[\boldsymbol{\omega}]_{-1\ 1} + |2_{\hat{\jmath}},\ 1_{\hat{J}_z}\rangle\, \mathbf{D}^{(2)}[\boldsymbol{\omega}]_{0\ 1} + |2_{\hat{\jmath}},\ 1_{\hat{J}_z}\rangle\, \mathbf{D}^{(2)}[\boldsymbol{\omega}]_{1\ 1} + |2_{\hat{\jmath}},\ 1_{\hat{J}_z}\rangle\, \mathbf{D}^{(2)}[\boldsymbol{\omega}]_{2\ 1}$$

Observe that some of the indexed objects have "negative indices". For half integer angular momenta, we obtain the following.

```
In[28]:= P[ω] · |(3/2)_ĵ, (1/2)_Ĵz⟩
```

$$\text{Out[28]}= \left|\left(\frac{3}{2}\right)_{\hat{\jmath}},\ \left(\frac{1}{2}\right)_{\hat{J}_z}\right\rangle \mathbf{D}^{(\frac{3}{2})}[\boldsymbol{\omega}]_{-\frac{3}{2}\ \frac{1}{2}} + \left|\left(\frac{3}{2}\right)_{\hat{\jmath}},\ \left(\frac{1}{2}\right)_{\hat{J}_z}\right\rangle \mathbf{D}^{(\frac{3}{2})}[\boldsymbol{\omega}]_{-\frac{1}{2}\ \frac{1}{2}} + \left|\left(\frac{3}{2}\right)_{\hat{\jmath}},\ \left(\frac{1}{2}\right)_{\hat{J}_z}\right\rangle \mathbf{D}^{(\frac{3}{2})}[\boldsymbol{\omega}]_{\frac{1}{2}\ \frac{1}{2}} + \left|\left(\frac{3}{2}\right)_{\hat{\jmath}},\ \left(\frac{1}{2}\right)_{\hat{J}_z}\right\rangle \mathbf{D}^{(\frac{3}{2})}[\boldsymbol{\omega}]_{\frac{3}{2}\ \frac{1}{2}}$$

There are several notational variations on the above approach that we could have taken. One would have been to explicitly include the angular momentum state number as an index in the indicial expression, as in $\mathcal{D}[\omega]^{j}{}_{nm}$. We will revisit angular momentum in quantum mechanics later, in §5.5 *Example: Angular Momentum*.

Actually, even excluding negative indices, many programs insist that 0 is not used as an index either, since in this case `-0` auto reduces to `0`, so all covariant zeros automatically become contravariant zeros. Again, this is disastrous, and the steps that certain programs take in order to avoid `-0` simplifying to `0` are quite contorted — see, for instance, CANTENS[52]).

In any case, the original point is clear: we need proper handling of indices, including negative indices and the index 0, if we are going to use tensors and, more generally, indexed objects in a consistent way throughout the physical sciences. In the next subsection, we introduce some semantics for handling indexed objects.

3.5.5 Simple Cartesian Vector Calculus I

Now that we can at least express our relations in a nice way, reindex expressions, and perform "tensorial assignments", we can present an extremely simplistic example involving this machinery. As good a choice as any is a simplistic treatment of a portion of Cartesian vector calculus. We will not do any "significant" calculations in this subsection, just simple calculations to describe the functioning of the notation.

In later chapters — §4 *Language Modifications* and others — we will make extensive changes to the underlying language. However, at this juncture, we restrict ourselves to using just standard *Mathematica* (including the *Notation* package). With this restriction, typically, our system would be implemented by a set of rewrite rules, which we would then apply or directly assign. Let us adopt such an approach.

In vector calculus, both the divergence and the curl of a vector field are valid operations. Thus, we introduce the following notations in the same way as was previously done in §2.4.1 *The Option Working Form*.

```
In[29]:= Notation[∇ . expr_ ⇔ Div[expr_]]
         Notation[∇ × expr_ ⇔ Curl[expr_]]
         Notation[v_⃗ ⇔ Vector[v_]]
```

For simplicity, in this subsection we restrict ourselves just to this toy segment of Cartesian vector calculus. Thus, all calculations we do here involve just constants (i.e., numbers), vectors, and/or the divergence and curl operators. Since we are going to expand our vector calculus formulas in terms partial derivative operators, let us consequently introduce a partial derivative operator notation.

```
In[32]:= Notation[∂̂ ⇔ operatorPD]
```

Here is the full form of a typical expression.

```
In[33]:= ∇ × ∇ × v̂ // FullForm
Out[33]//FullForm=
Curl[Curl[Vector[v]]]
```

For further progress, we need the `NonCommutativeTimes` operation. It was previously introduced in §2.7.4 *Example Calculations from Physics;* but for convenience, we re-introduce it here.

```
In[34]:= ConstQ[a_?NumberQ] := True
         ConstQ[_] := False;

In[36]:= InfixNotation[·, NonCommutativeTimes];
         SetAttributes[NonCommutativeTimes, {Flat, OneIdentity}];

In[37]:= ℓ___ · (c_?ConstQ b_) · r___ := cℓ · b · r /; ({ℓ, r} =!= {})
         ℓ___ · c_?ConstQ · r__ := cℓ · r
         ℓ__ · c_?ConstQ · r___ := cℓ · r
         ℓ___ · (a_ + b_) · r___ := ℓ · a · r + ℓ · b · r
```

We must also declare that our `NonCommutativeTimes` is a tensorial multiplicative head. This enables `DummyIndices` and several of the other functions defined in the *Tensors* package to work correctly across the head `NonCommutativeTimes`.

```
In[41]:= DeclareTensorialMultiplicativeHead[
             NonCommutativeTimes, Commutivity → False]
```

Now we can codify our rules for `Div` and `Curl`.

```
In[42]:= ∇ . v_ :≂ ∂̂_j · v_j

         ∇ × v_i_ :≂ ϵ_ijk · ∂̂_j · v_k
```

Here we have used the Levi-Civita tensor defined as follows.

$$\epsilon_{ijk} = \begin{cases} +1 & \text{if } \{i, j, k\} \text{ are an even permutation of } 1, 2, 3 \\ 0 & \text{if any of the } i, j, k \text{ are not unique} \\ -1 & \text{if } \{i, j, k\} \text{ are an odd permutation of } 1, 2, 3 \end{cases}$$

Actually, the Levi-Civita tensor is a tensor density, but we will not enter into these nuances at this stage — see d'Inverno [81] or Weinberg[333] for details. The Levi-Civita tensor is extremely simple to implement.

```
In[44]:= ϵ_i_Integer_j_Integer_k_Integer := Signature @ {i, j, k}
```

We must also include the fact that the Levi-Civita tensor is a constant.

```
In[45]:= ConstQ @ ϵ___ = True;
```

Simple calculations can now be entered. Let us use our tools to prove the extremely well known result that $\nabla . \nabla \times anything = 0$.

```
In[46]:= ∇ . ∇ × v⃗ // ReIndex
```

$$\text{Out[46]=}\ \hat{\partial}_a \cdot \hat{\partial}_b \cdot \vec{v}_c\, \epsilon_{abc}$$

Our canonicalization routines, which we later develop in §6 *An Algorithm for Tensor Simplification*, will trivially simplify this expression to zero. However, at present, our routines can not yet recognize that a product of a symmetric object and antisymmetric object is zero. Consequently, we resort to the more brutish solution of *contracting* the summed indices over the actual coordinates. It is extremely easy to implement a function, `Contract`, which contracts the dummy indices in a tensorial expression. (Later, in §7.2 *Tensor Manipulations and Tensor Calculus*, `Contract` will be adapted to contracting over specific index classes.)

```
In[47]:= Contract @ expr_Times := SumOverIndices[expr, DummyIndices @ expr]
         SumOverIndices[expr_, indices_] :=
           With[{iterators = Sequence @@ ({#, 1, 3} & /@ indices)}, Sum[expr, iterators]]
```

Here is a simple example of a contraction.

```
In[49]:= Contract[T_a A_ab]
```

$$\text{Out[49]=}\ \mathbf{A}_{1b}\mathbf{T}_1 + \mathbf{A}_{2b}\mathbf{T}_2 + \mathbf{A}_{3b}\mathbf{T}_3$$

Applying `Contract` to $\nabla.\nabla\times\vec{v}$ yields

```
In[50]:= Contract[∇ . ∇ × v⃗]
```

$$\text{Out[50]=}\ \hat{\partial}_1 \cdot \hat{\partial}_2 \cdot \vec{v}_3 - \hat{\partial}_1 \cdot \hat{\partial}_3 \cdot \vec{v}_2 - \hat{\partial}_2 \cdot \hat{\partial}_1 \cdot \vec{v}_3 + \hat{\partial}_2 \cdot \hat{\partial}_3 \cdot \vec{v}_1 + \hat{\partial}_3 \cdot \hat{\partial}_1 \cdot \vec{v}_2 - \hat{\partial}_3 \cdot \hat{\partial}_2 \cdot \vec{v}_1$$

The partial derivative operators commute, that is, $\hat{\partial}_a \cdot \hat{\partial}_b == \hat{\partial}_b \cdot \hat{\partial}_a$. Consequently, we can order the derivatives according to their indices. This provides a "normal ordering" of a sequence of these operators. For elegance, we specify the commutation rule in terms of the following notation, which lexicographically compares one index with another.

```
In[51]:= Notation[a_ <tex b_ ⟹ Not[OrderedQ[{b_, a_}]]]
```

```
In[52]:= ∂̂_b_ · ∂̂_a_ := ∂̂_a · ∂̂_b /; a <tex b
```

This results in the derivative operators being in "normal order", as the following example demonstrates.

```
In[53]:= ∂̂_1 · ∂̂_0 · v⃗_2 - ∂̂_b · ∂̂_a · ∂̂_d · ∂̂_b · v⃗_3
```

$$\text{Out[53]=}\ \hat{\partial}_0 \cdot \hat{\partial}_1 \cdot \vec{v}_2 - \hat{\partial}_a \cdot \hat{\partial}_b \cdot \hat{\partial}_b \cdot \hat{\partial}_d \cdot \vec{v}_3$$

With normal ordering of derivatives now active, we repeat the contraction of the expression for $\nabla.\nabla\times\vec{v}$.

```
In[54]:= ∇ . ∇ × v⃗ // Contract
```

```
Out[54]= 0
```

Thus, as we knew all along, $\nabla.\nabla\times\vec{v}$ is identically zero. Later, we will be able to show this just by canonicalizing $\hat{\partial}_a \cdot \hat{\partial}_b \cdot \vec{v}_c \epsilon_{abc}$. Of greater importance is the fact that performing such canonicalizations is *much* faster than performing contractions and manipulating the larger expressions resulting from having the actual coordinate indices present in the expressions. (The latter is the way Sheep[84] operates.) The simple contraction of $\hat{\partial}_a \cdot \hat{\partial}_b \cdot \vec{v}_c \epsilon_{abc}$ has 27 terms in it before the resolution of the Levi-Civita tensor. Even after this resolution, the expansion has 6 terms. The contraction $\hat{\partial}_a \cdot \hat{\partial}_b \cdot \vec{v}_d \epsilon_{cbd} \epsilon_{iac}$, which we encounter in our next example, has 81 terms before the resolution of the Levi-Civita tensor. If at all possible, it is better to deal with the single compact symbolic term.

3.5.6 Simple Cartesian Vector Calculus II

Let us now turn to a slightly different example. Our next goal is to compute the i^{th} component of $\nabla\times\nabla\times\vec{v}$.

```
In[55]:= (∇ × ∇ × v⃗)_i // ReIndex
```

```
Out[55]= ∂̂_a · ∂̂_b · v⃗_d ϵ_cbd ϵ_iac
```

We could contract the summed indices, but this would lead to a semi–ugly expression that would not really be all that illuminating. Instead, let us use the fact that the product of two Levi-Civita tensors can be expanded in terms of Kronecker delta functions ([81, 326, 333]). Specifically,

$$\epsilon_{ijk}\epsilon_{lmn} = \begin{vmatrix} \delta_{il} & \delta_{im} & \delta_{in} \\ \delta_{jl} & \delta_{jm} & \delta_{jn} \\ \delta_{kl} & \delta_{km} & \delta_{kn} \end{vmatrix} \tag{3.5.e}$$

This determinant can be easily calculated.

```
In[56]:= Det @ ( δ_il  δ_im  δ_in )
               ( δ_jl  δ_jm  δ_jn )
               ( δ_kl  δ_km  δ_kn )
```

```
Out[56]= -δ_in δ_jm δ_kl + δ_im δ_jn δ_kl + δ_in δ_jl δ_km -
         δ_il δ_jn δ_km - δ_im δ_jl δ_kn + δ_il δ_jm δ_kn
```

Let us embody the above expansion in the following simplification rule. (We restrict the application of this rule to only those cases where some reduction is possible.)

```
In[57]:= ϵSimplification = ϵ_{i_ j_ k_} ϵ_{l_ m_ n_} :>
           (-δ_{i n} δ_{j m} δ_{k l} + δ_{i m} δ_{j n} δ_{k l} + δ_{i n} δ_{j l} δ_{k m} - δ_{i l} δ_{j n} δ_{k m} -
             δ_{i m} δ_{j l} δ_{k n} + δ_{i l} δ_{j m} δ_{k n}) /; {i, j, k} ∩ {l, m, n} ≠ {}
```

Applying this rule to our calculation, we obtain the following.

```
In[58]:= ((∇ × ∇ × v⃗)_i // ReIndex) /. ϵSimplification

Out[58]= ∂̂_a · ∂̂_b · v⃗_d (δ_{b i} δ_{c c} δ_{d a} - δ_{b c} δ_{c i} δ_{d a} -
             δ_{b i} δ_{c a} δ_{d c} + δ_{b a} δ_{c i} δ_{d c} + δ_{b c} δ_{c a} δ_{d i} - δ_{b a} δ_{c c} δ_{d i})
```

```
In[59]:= Expand @ %

Out[59]= ∂̂_a · ∂̂_b · v⃗_d δ_{b i} δ_{c c} δ_{d a} - ∂̂_a · ∂̂_b · v⃗_d δ_{b c} δ_{c i} δ_{d a} -
           ∂̂_a · ∂̂_b · v⃗_d δ_{b i} δ_{c a} δ_{d c} + ∂̂_a · ∂̂_b · v⃗_d δ_{b a} δ_{c i} δ_{d c} +
           ∂̂_a · ∂̂_b · v⃗_d δ_{b c} δ_{c a} δ_{d i} - ∂̂_a · ∂̂_b · v⃗_d δ_{b a} δ_{c c} δ_{d i}
```

In order to simplify this expression, we need to create some rules for the simplification of expressions involving the Kronecker delta function.

```
In[60]:= δSimplification =
           {δ_{i_Integer j_Integer} :> If[i ≡ j, 1, 0, 0], δ_{a_ a_} → 3,
            (δ_{a_ b_} T_) /; Not @ FreeQ[T, a] :> (T /. a → b),
            (δ_{a_ b_} T_) /; Not @ FreeQ[T, b] :> (T /. b → a)};
```

Applying these new Kronecker delta simplification rules to our last result, we obtain the following.

```
In[61]:= %% //. δSimplification

Out[61]= -(∂̂_a · ∂̂_a · v⃗_i) + ∂̂_a · ∂̂_i · v⃗_a
```

This shows that $\nabla \times \nabla \times \vec{v} = \nabla(\nabla . \vec{v}) - \nabla^2 \vec{v}$, which of course confirms this standard identity.

Let us perform one final example and calculate $\nabla \times \vec{u} \times \vec{v}$ (which groups as $\nabla \times (\vec{u} \times \vec{v})$). We do this by first adding yet another rule. Specifically, we define the cross product of two vectors as follows.

```
In[62]:= u⃗_ × v⃗_ i_ :≔ ϵ_{i j k} · u⃗_j · v⃗_k
```

Now we calculate $\nabla \times \vec{u} \times \vec{v}$.

```
In[63]:= ((∇ × ū × v̄)_i /. εSimplification // Expand) //. δSimplification) //
           ReIndex

Out[63]= -(∂̂_a · ū_a · v̄_i) + ∂̂_a · ū_i · v̄_a
```

By inspection, we see the result is $(\nabla.\vec{v})\vec{u} - (\nabla.\vec{u})\vec{v}$. This expression further reduces if we introduce yet more rules. However, we have given the general idea of how to proceed.

It should be noted that the approach we gave to Cartesian vector calculus was developmental and possibly piecemeal. Alternatively, we could have just given the final set of rules to be obeyed and then performed the calculations. But this would have had the drawback of not being very illustrative. The approach we have taken, via using components, is of course not the only way to perform vector calculus calculations. Indeed, one could implement the relations more generally in a set of rewriting rules that did not make use of coordinates. Such an approach is taken in the package VECTAN by Fidler[109] or GeneralVectorAnalysis by Qin[270, 271]. (Actually the latter author uses the notation package for defining his notations.) Nevertheless, in practice, component calculations arise in many facets of physics, and the toy calculations we have performed in this subsection should make the approach we later take more familiar. In any case, even the approach of manipulating expressions containing divs and curls without reducing expressions to components, would be greatly aided by the notations we have introduced in this chapter.

3.5.7 Extended Tensor Syntax

Actually, there is yet one further addition to the notation for tensors which is required. For motivation, consider that some tensorial definitions can be used with both contravariant and covariant indices. For instance, the expansion of the Weyl tensor in terms of the Riemann, Ricci, and metric tensors is the following — see [81, 255, 326, 333].

$$C_{\alpha\beta\gamma\delta} \longrightarrow \frac{R^{\mu}{}_{\mu}\left(-g_{\alpha\delta}g_{\beta\gamma} + g_{\alpha\gamma}g_{\beta\delta}\right)}{(d-2)(d-1)} - \frac{g_{\beta\delta}R_{\alpha\gamma} - g_{\beta\gamma}R_{\alpha\delta} - g_{\alpha\delta}R_{\beta\gamma} - g_{\alpha\gamma}R_{\beta\delta}}{d-2} + R_{\alpha\beta\gamma\delta}$$

However, since all of the components involved are real tensors and no explicit derivatives appear in the formula, it transpires that we can raise and lower any of the indices at our discretion. How then can we introduce a single formula that will capture all 16 different combinations of elevations, that is ↑↑↑↑,↑↑↑↓, ↑↑↓↑, ↑↑↓↓, ..., ↓↓↓↓? We need definitions that maintain the elevation of their given indices but accept indices of any elevation.

The extension is straightforward. Notationally, we will use starred indices in a subscript position to denote indices of indeterminate elevation. To accomodate this extension, we of course need to modify both the parsing and the formatting behavior of our tensorial notation. First, for formatting, we need to format any tensor with an index that is neither high nor low as a low starred index. We consequently add a single rule to those already given for `makeStringIndexPair` in §3.4.4 *Definitions for Tensor Formatting*.

```
In[64]:= makeStringIndexPair @ index_ :=
           {"", SuperscriptBox[MakeBoxes @ index,
                StyleBox["*", "TensorStarStyle"]]}
```

We must also state that all indices are valid, not just the high and low ones.

```
In[65]:= Clear @ validIndicesQ;
         validIndicesQ @ _ = True;
```

With these modifications, tensors like the following are now formatted as shown.

```
In[67]:= Tensor[T, {a, b*}]
```

$$\text{Out[67]=}\quad \mathbf{T}_{a}^{\ b}$$

In order to reciprocally modify parsing, we need to change how `parseStringIndices` functions. We clear the old definition and add a specialized case.

```
In[68]:= Clear @ parseStringIndices;
```

```
In[69]:= parseStringIndices @
           {_?whiteSpaceQ, SuperscriptBox[index_?validStringIndexQ,
                "*" | StyleBox["*", "TensorStarStyle"]]} :=
         parseSpecial @ index;
       parseStringIndices @
           {SuperscriptBox[index_?validStringIndexQ,
                "*" | StyleBox["*", "TensorStarStyle"]],
            _?whiteSpaceQ} := parseSpecial @ index;
```

If the index pair to be parsed is not a "starred" index, then we must fall back to our old parsing behavior.

```
In[71]:= parseStringIndices @ {_?whiteSpaceQ, index_?validStringIndexQ} :=
           RowBox @ {"Low", "[", parseSpecial @ index, "]"}
         parseStringIndices @ {index_?validStringIndexQ, _?whiteSpaceQ} :=
           RowBox[{"High", "[", parseSpecial@index, "]"}]
         parseStringIndices @ other_ :=
           RowBox @ {"tensorParseError", "[", other, "]"}
```

Technical Note: Actually, the complete code for the *Tensors`* package contains a few additional lines of code. These ensure that nested tensors parse correctly, as well as ensuring the tensor "name" is parenthesized if it is a composite expression as opposed to a symbol, for instance, something like `Tensor[A+B, {High[a]}]`.

Let us demonstrate that our new parsing behavior is working as desired.

```
In[74]:= T^{a b*}_{c d*} // FullForm
```

Out[74]//FullForm=

```
Tensor[T, List[High[a], b, Low[c], d]]
```

We can immediately use our new parsing behavior to define the Weyl tensor with indices of any elevation in terms of the Riemann, Ricci and metric tensors.

In[75]:=
$$C_{\alpha\_\ \beta\_\ \gamma\_\ \delta\_} := \left(\left(-\mathbf{g}_{\alpha^{\cdot}\ \delta^{\cdot}}\ \mathbf{g}_{\beta^{\cdot}\ \gamma^{\cdot}} + \mathbf{g}_{\alpha^{\cdot}\ \gamma^{\cdot}}\ \mathbf{g}_{\beta^{\cdot}\ \delta^{\cdot}}\right)\mathbf{R}^{\mu}{}_{\mu}\right) / ((d-2)\ (d-1)) - \frac{1}{d-2}$$
$$\left(\mathbf{g}_{\beta^{\cdot}\ \delta^{\cdot}}\mathbf{R}_{\alpha^{\cdot}\ \gamma^{\cdot}} - \mathbf{g}_{\beta^{\cdot}\ \gamma^{\cdot}}\mathbf{R}_{\alpha^{\cdot}\ \delta^{\cdot}} - \mathbf{g}_{\alpha^{\cdot}\ \delta^{\cdot}}\mathbf{R}_{\beta^{\cdot}\ \gamma^{\cdot}} - \mathbf{g}_{\alpha^{\cdot}\ \gamma^{\cdot}}\mathbf{R}_{\beta^{\cdot}\ \delta^{\cdot}}\right) + \mathbf{R}_{\alpha^{\cdot}\ \beta^{\cdot}\ \gamma^{\cdot}\ \delta^{\cdot}}$$

Here is a Weyl tensor demonstrating that our formula works with indices of any elevation.

In[76]:=
$$C_{\alpha}{}^{\beta}{}_{\gamma}{}^{\delta}$$

Out[76]=
$$\frac{\left(-\mathbf{g}^{\beta}{}_{\gamma}\,\mathbf{g}_{\alpha}{}^{\delta} + \mathbf{g}^{\beta\,\delta}\,\mathbf{g}_{\alpha\,\gamma}\right)\mathbf{R}^{\mu\$1538}{}_{\mu\$1538}}{(-2+d)\ (-1+d)} -$$
$$\frac{-\mathbf{g}_{\alpha\,\gamma}\mathbf{R}^{\beta\,\delta} - \mathbf{g}_{\alpha}{}^{\delta}\mathbf{R}^{\beta}{}_{\gamma} - \mathbf{g}^{\beta}{}_{\gamma}\mathbf{R}_{\alpha}{}^{\delta} + \mathbf{g}^{\beta\,\delta}\mathbf{R}_{\alpha\,\gamma}}{-2+d} + \mathbf{R}_{\alpha}{}^{\beta}{}_{\gamma}{}^{\delta}$$

As a further example using this notation, we will calculate the complete contraction in d dimensions of the Weyl tensor $C^{\alpha\beta\gamma\delta}\,C_{\alpha\beta\gamma\delta}$ in §7 *Tensor Calculus, Applications, and Quasi-Spin*, where we will have at our disposal further machinery for canonicalization. (Actually, we could perform this contraction at the present stage, but we could not simplify the result.)

Now that we have introduced all the notations for this section, we can progress onto some final examples using the new extended tensorial notation. In particular, we consider the relatively simple examples of metric contractions and generalized Kronecker delta symbols.

3.5.8 Examples Using the Extended Tensor Syntax

This subsection endeavors to illustrate the extended tensor syntax introduced in the previous subsection. One of the great benefits of the extended tensorial notation is that we can now create rules for tensors with indices specified by patterns like $\ell\_\_^{*}$. For instance, if we want to contract the metric tensor with other tensors, we can achieve this using the following set of rules.

In[77]:=
$$\mathbf{g}^{a\_\ b\_}\,\mathbf{T}_{\_\,\ell\_\_^{\cdot}\ a\_\ r\_\_^{\cdot}} :\to \mathbf{T}_{\ell^{\cdot}\ r^{\cdot}}{}^{b}\ /;\ !\,\text{IntegerQ}[a],$$
$$\mathbf{g}^{a\_\ b\_}\,\mathbf{T}_{\_\,\ell\_\_^{\cdot}\ b\_\ r\_\_^{\cdot}} :\to \mathbf{T}_{\ell^{\cdot}\ r^{\cdot}}{}^{a}\ /;\ !\,\text{IntegerQ}[a],$$
$$\mathbf{g}_{a\_\ b\_}\,\mathbf{T}_{\_\,\ell\_\_^{\cdot}}{}^{a\_}{}_{r\_\_^{\cdot}} :\to \mathbf{T}_{\ell^{\cdot}\ b\,r^{\cdot}}\ /;\ !\,\text{IntegerQ}[a],$$
$$\mathbf{g}_{a\_\ b\_}\,\mathbf{T}_{\_\,\ell\_\_^{\cdot}}{}^{b\_}{}_{r\_\_^{\cdot}} :\to \mathbf{T}_{\ell^{\cdot}\ a\,r^{\cdot}}\ /;\ !\,\text{IntegerQ}[a]\};$$

These rules, `ContractMetricRules`, assume that we are working in a locally flat space. We could easily eliminate this restriction by putting in a few simple conditions to make the rules more general, but we are only trying to illustrate the functioning of the rules at this stage. In this set of rules, we test to ensure that a is not an integer, that is, it is not already a fixed coordinate. This is because, for example, $g^{a0} T^{b}{}_{0} \neq T^{ba}$, but it is always true that $g^{ac} T^{b}{}_{c} = T^{ba}$ if T is tensorial, etc.

In addition to the above contraction rules, the metric tensor with indices of any elevation is symmetric, thus we should "normal order" its indices.

```
In[78]:= g /: g_b_* a_* := g a* b* /; a <tex b
```

Here is an example demonstrating the contractions of the metric tensor with other tensors.

```
In[79]:= g^ϵ ζ g_α γ g_β ϵ g_δ ζ R^α α / ((d - 2) (d - 1)) //. ContractMetricRules
```

$$\text{Out[79]=}\quad \frac{g_{\beta\delta} R_{\gamma\alpha}}{(-2+d)\ (-1+d)}$$

Actually, as a consistency check, let us perform another calculation using the contraction of our metrics. In the previous subsection, §3.5.7 *Extended Tensor Syntax*, we gave a definition of the Weyl tensor that functioned for indices with any elevation. Let us check that raising and lowering the indices using the metric tensor indeed functions correctly. If our definitions are functioning correctly, it must be the case that $C_{\alpha\beta\gamma\delta} = g_{\nu\delta} g_{\mu\beta} C_{\alpha}{}^{\mu}{}_{\gamma}{}^{\nu}$. We can show this as follows.

```
In[80]:= C_α β γ δ - g_ν δ g_μ β C_α^μ_γ^ν // Expand // ReIndex
```

$$\begin{aligned}\text{Out[80]=}\quad & -\frac{g_{\alpha\gamma} g_{\beta\epsilon} g_{\delta\zeta} R^{\epsilon\zeta}}{-2+d} - \frac{g^{\zeta}{}_{\alpha} g_{\beta\epsilon} g_{\delta\zeta} R^{\epsilon}{}_{\gamma}}{-2+d} - \frac{g_{\alpha\delta} g_{\beta\gamma} R^{\epsilon}{}_{\epsilon}}{(-2+d)\ (-1+d)} + \\ & \frac{g_{\alpha\gamma} g_{\beta\delta} R^{\epsilon}{}_{\epsilon}}{(-2+d)\ (-1+d)} + \frac{g^{\epsilon}{}_{\gamma} g^{\eta}{}_{\alpha} g_{\beta\epsilon} g_{\delta\eta} R^{\zeta}{}_{\zeta}}{(-2+d)\ (-1+d)} - \frac{g^{\epsilon\eta} g_{\alpha\gamma} g_{\beta\epsilon} g_{\delta\eta} R^{\zeta}{}_{\zeta}}{(-2+d)\ (-1+d)} - \\ & \frac{g^{\epsilon}{}_{\gamma} g_{\beta\epsilon} g_{\delta\zeta} R_{\alpha}{}^{\zeta}}{-2+d} - \frac{g_{\beta\delta} R_{\alpha\gamma}}{-2+d} + \frac{g^{\epsilon\zeta} g_{\beta\epsilon} g_{\delta\zeta} R_{\alpha\gamma}}{-2+d} + \\ & \frac{g_{\beta\gamma} R_{\alpha\delta}}{-2+d} + \frac{g_{\alpha\delta} R_{\beta\gamma}}{-2+d} + \frac{g_{\alpha\gamma} R_{\beta\delta}}{-2+d} - g_{\beta\epsilon} g_{\delta\zeta} R_{\alpha}{}^{\epsilon}{}_{\gamma}{}^{\zeta} + R_{\alpha\beta\gamma\delta}\end{aligned}$$

It now remains to contract all of the metrics present in the above expression.

```
In[81]:= % //. ContractMetricRules
```

$$\text{Out[81]=}\quad -\frac{g_{\alpha\delta} g_{\beta\gamma} R^{\epsilon}{}_{\epsilon}}{(-2+d)\ (-1+d)} + \frac{g_{\alpha\gamma} g_{\beta\delta} R^{\epsilon}{}_{\epsilon}}{(-2+d)\ (-1+d)} + \frac{g_{\alpha\delta} g_{\beta\gamma} R^{\zeta}{}_{\zeta}}{(-2+d)\ (-1+d)} - \frac{g_{\alpha\gamma} g_{\beta\delta} R^{\zeta}{}_{\zeta}}{(-2+d)\ (-1+d)}$$

```
In[82]:= % // ReIndex

Out[82]= 0
```

We thus see that indeed our formula for the raising and lowering of the metric tensor is consistent with our definition of the Weyl tensor with indices of any elevation. (In order to make our metric contraction rules function correctly in non-locally flat spaces, we can just attach conditions to our rules to ensure that the indices we are trying to raise do not occur in tensorial objects having partial derivatives. We elaborate on this in §7.3.1 *Standard Tensors in General Relativity*.)

In §3.5.6 *Simple Cartesian Vector Calculus II*, we considered the product of two 3-dimensional Levi-Civita tensors in terms of the Kronecker delta function. We would now like to generalize this to the case of the product of two d-dimensional Levi-Civita tensors. We will denote this product by the generalized Kronecker delta function, that is, $\delta^{i\,j\,\ldots\,k}{}_{l\,m\,\ldots\,n} = \epsilon^{i\,j\,\ldots\,k}\,\epsilon_{l\,m\,\ldots\,n}$. The same general form of a determinant expansion used in 3 dimensions applies to this product in d dimensions. That is, we can express the product of two d-dimensional Levi-Civita tensors in terms of a $d \times d$ determinant, as follows.

$$\epsilon^{i\,j\,\ldots\,k}\,\epsilon_{l\,m\,\ldots\,n} = \delta^{i\,j\,\ldots\,k}{}_{l\,m\,\ldots\,n} = \begin{vmatrix} \delta^{i}{}_{l} & \delta^{i}{}_{m} & \cdots & \delta^{i}{}_{n} \\ \delta^{j}{}_{l} & \delta^{j}{}_{m} & \cdots & \delta^{j}{}_{n} \\ \vdots & \vdots & \ddots & \vdots \\ \delta^{k}{}_{l} & \delta^{k}{}_{m} & \cdots & \delta^{k}{}_{n} \end{vmatrix} \tag{3.5.f}$$

We could blithely just expand all generalized Kronecker delta's in terms of standard Kronecker delta's. This is the approach we previously took in §3.5.6 *Simple Cartesian Vector Calculus II*, and a routine to carry out such an expansion in the d-dimensional case is easily given.

```
In[83]:= Notation[ℓ_fen ⟺ Length[ℓ_]];
         Unprotect @ δ;

In[85]:= Clear @ δ

In[86]:= δ /: δ indices__ := With[{ℓ = {indices}fen / 2},
              Det @ Outer[δ i_, j_ &i,j, Sequence @@ Partition[{indices}, ℓ]] /;
                ℓ ≥ 2 ∧ IntegerQ @ ℓ]
```

Here is an example testing this expansion.

```
In[87]:= δ i j k
         i l m

Out[87]= -δ i δ j δ k + δ i δ j δ k + δ i δ j δ k -
           m   l   i   l   m   i   m   i   l

           δ i δ j δ k - δ i δ j δ k + δ i δ j δ k
           i   m   l   l   i   m   i   i   l   m
```

However, to simplify these results, we need to again introduce some simplification rules for Kronecker delta functions.

```
In[88]:= δSimplification = {
           δ^(i_Integer)_(j_Integer) :> If[i == j, 1, 0, 0],
           δ^(a_)_(a_) :> d,
           δ^(b_)_(a_) δ^(a_)_(c_) :> δ^b_c /; ! IntegerQ[a],
           δ^(a_)_(b_) δ^(c_)_(a_) :> δ^c_b /; ! IntegerQ[a],
           δ_(b_*) _(a_*) :> δ_(a* b*) /; a <lex b};

In[89]:= δ^(i j k)_(i l m) //. δSimplification // Simplify

Out[89]= -(-2 + d) (δ^j_m δ^k_l - δ^j_l δ^k_m)
```

This approach is only partially adequate. It takes some time to simplify generalized Kronecker delta functions like the following.

```
In[90]:= δ^(i j k l p q)_(i j k l a b) //. δSimplification // Simplify // Timing

Out[90]= {Null Second, -(120 - 154 d + 71 d^2 - 14 d^3 + d^4) (δ^p_b δ^q_a - δ^p_a δ^q_b)}
```

Consequently, we would like an approach the does not automatically expand out all generalized Kronecker delta's into sums of products of standard Kronecker delta's, but yet puts them into a "standard form".

Unfortunately, we do not actually introduce our canonicalization functions until §6 *An Algorithm for Tensor Simplification*, and only discuss the application of such algorithms in §7 *Tensor Calculus, Applications, and Quasi-Spin*. So until then, protracted discussions involving such algorithms are delayed. However, once we introduce the canonicalization functions, then all we have to do is declare that the generalized Kronecker delta has the following symmetries.

$$\begin{aligned}
\delta^{h_1 \ldots i\, j \ldots h_n}_{l_1 \ldots l_n} &= -\delta^{h_1 \ldots j\, i \ldots h_n}_{l_1 \ldots l_n} \\
\delta^{h_1 \ldots h_n}_{l_1 \ldots i\, j \ldots l_n} &= -\delta^{h_1 \ldots h_n}_{l_1 \ldots j\, i \ldots l_n} \\
\delta^{h_1 \ldots h_n}_{l_1 \ldots l_n} &= \delta_{l_1 \ldots l_n}^{h_1 \ldots h_n}
\end{aligned} \tag{3.5.g}$$

Additionally, note that the generalized Kronecker delta computationally reduces to a more simplified version whenever it has a summed index present.

$$\delta^{i\, h_1 \ldots h_n}_{i\, l_1 \ldots l_n} \longrightarrow (d-n)\,\delta^{h_1 \ldots h_n}_{l_1 \ldots l_n} \tag{3.5.h}$$

In the above, d is the number of dimensions in our base space. The identities (3.5.g) follow trivially from the properties of the determinant. Only the rule (3.5.h) is non-trivial to show. It can be found by experimenting with the implementation of the generalized Kronecker delta we have given; alternatively, it can be shown to be true by other arguments. However, for the present, since we cannot simply specify the symmetries of our generalized Kronecker delta function and leave the rest to the canonicalization algorithm, we must instead create our own rules which achieve (3.5.g) and (3.5.h). Here is one such implementation.

```
In[91]:= Clear @ δ

In[92]:= δ /: δ_{a___*, j_*, i_*, ℓ___*} := -δ_{a*, i*, j*, ℓ*} /; (i <_lex j ∧ Abs[{ℓ}_len - {a}_len] > 1);

          δ /: δ_{a___*, i_, b___*, i_, c___*} := With[{ℓ = {a}_len, m = {b}_len, r = {c}_len},

                kroneckerMultiplier[ℓ, m, r] δ_{a*, b*, c*} /;

                  ! IntegerQ[/] ∧ EvenQ[ℓ + m + r]];

          δ /: δ_{a___*, i_, m___*, i_, ℓ___*} := With[{ℓ = {a}_len, m = {b}_len, r = {c}_len},

                kroneckerMultiplier[ℓ, m, r] δ_{a*, b*, c*} /;

                  ! IntegerQ[/] ∧ EvenQ[ℓ + m + r]];

In[95]:= kroneckerMultiplier[ℓ_, m_, r_] :=
           With[{n = (ℓ + m + r) / 2}, 0 /; ℓ > 1 + n ∨ r > 1 + n]
         kroneckerMultiplier[ℓ_, m_, r_] :=
           With[{n = (ℓ + m + r) / 2}, (d - n) (-1)^ℓ (-1)^(n-r)]
```

Here are a few calculations illustrating the identities (3.5.g) and (3.5.h).

```
In[97]:= δ^{i j k p}_{p m i l}

Out[97]= -(-3 + d) (-2 + d) δ^{j k}_{l m}

In[98]:= δ^{i j k}_{i l m} == -δ^{i j k}_{i m l}

Out[98]= True

In[99]:= δ^{i j k}_{i m l} == (d - 2) δ^{j k}_{m l}

Out[99]= True
```

We can compare the timings for this newer method to our previous method and confirm that it is *much* faster.

```
In[100]:= δ^{i j k l p q}_{i j k l a b} // Timing

Out[100]= {Null Second, (-5 + d) (-4 + d) (-3 + d) (-2 + d) δ^{p q}_{a b}}
```

Necessarily, this answer must be the same as that returned by our original method. This can be corroborated by turning back to our original calculation of this generalized Kronecker delta, and confirming that the original coefficient was $(120 - 154\,d + 71\,d^2 - 14\,d^3 + d^4)$, which equals our new coefficient.

```
In[101]:= Expand[(-5 + d) (-4 + d) (-3 + d) (-2 + d)] == (120 - 154 d + 71 d^2 - 14 d^3 + d^4)

Out[101]= True
```

We could actually do this whole calculation in a faster but less intuitive way. Basically, we could find all the dummy indices and remove them, multiplying by the signatures necessary when removing them. We would do this as follows.

```
In[102]:= Clear @ δ

In[103]:= δ /: δ_indices__ :=

            With[{ℓ = {indices}_ten / 2, dummies = Union @ DummyIndices[T_indices]},
              Module[{indices1, indices2, rest1, rest2, n, s1, s2, allDummies},
                {indices1, indices2} = Partition[{indices}, ℓ];
                n = dummies_ten;
                allDummies = Flatten @ {High /@ dummies, Low /@ dummies};
                {rest1, rest2} = {indices1 \ allDummies, indices2 \ allDummies};
                s1 = Signature[(indices1 ⋂ allDummies) +_{} rest1]
                     Signature[indices1];
                s2 = Signature[(indices2 ⋂ allDummies) +_{} rest2]
                     Signature[indices2];
                (Times @@ Table[d - j, {j, ℓ - n, ℓ - 1}]) * s1 *
                  s2 * Tensor[δ, rest1 +_{} rest2]] /;
              ℓ ≥ 2 ∧ IntegerQ @ ℓ ∧ dummies ≠ {}]

In[104]:= δ^{i j k l p q}_{i j k l a b} // Timing

Out[104]= {Null Second, (-5 + d) (-4 + d) (-3 + d) (-2 + d) δ^{p q}_{a b}}
```

It transpires that this last algorithm turns out to be faster than our second version for large numbers of indices but slower for small numbers of indices, like 4 to 8 indices. This last algorithm is very similar to that given in Trott[312].

Incidentally, similar sorts of symmetries arise in quantum mechanics. Specifically, the wave function of a multi-particle system consisting of fermionic particles must be anti-symmetric in the interchange of particles. This product wavefunction can be constructed using Slater determinants [55, 68, 30, 334, 206].

3.5.9 Concluding Remarks

The problem sets in this section were largely chosen so that they would be familiar to both mathematicians and physicists. It must be repeated that many of the calculations were not intended to provide the "best" way to solve the problems addressed. However, many of these same problems will be revisited, elaborated upon in detail and given adequate solutions in the coming chapters.

Instead of presenting simplistic solutions to "toy" problems in order to illustrate our tensorial notations, we could have taken the approach that is standard in computer science texts, namely to provide such illustrations in terms of generic functions like `foo` and `goo`. This has the advantage that it in no way advocates a method of solution that is non-optimal. However, it has the drawback that it involves expressions which look unfamiliar, or indeed, have no meaning outside the context of the concept being illustrated. (The author's personal preference is for the latter, but considering the intended reader, the former has been chosen.)

It should also be mentioned, yet again, that the computations in this section were clear and concise. We used the underlying notational system physicists work with. Our tensorial assignment operator is just a small step from normal assignment and elegantly handles the dummy index problem. This typifies the intermixing of notations and solutions to problems with the minimum intrusion upon normal working paradigms. The notation for tensors was demonstrated through many examples. We saw it seamlessly working with Cartesian, non-Cartesian and even just indexed objects, in definitions and in calculations. In short, it was simply "used", as opposed to being a problem to contend with.

It should be obvious that, with the tensorial objects, at least the tensorial syntax is functioning as an integral part of our working environment. It remains to furnish the tensorial objects with a correspondingly thorough and well designed semantics to complement the syntax. This is the goal of §7 *Tensor Calculus, Applications, and Quasi-Spin*.

3.6 Conclusions and the Future

3.6.1 Summary

We have now completed the first of our major goals. The setting and representations in which we choose to carry out our calculations can now be tailored to the specific notations of a given field. The functions in the *Notation* package automatically set up parsing and formatting rules for the specified notations, including the correct parenthesizing, styling, and precedences. In short, we can define notations.

It is true that we have not yet specified under what paradigm various computations or calculations will be carried out. (This shortcoming will be remedied in later chapters.) What we have accomplished at this stage is that we are now able to work and create definitions in the language in which we are used to working. Davenport[88], in his predictions and plans for the future, made the prognosis that it would just not be possible to have the input in the correct notations that one uses in output. He believed that it would be just too difficult. Indeed, as a point of interest, many developers within Wolfram Research, at the time I embarked upon the package, thought the task of creating the *Notation* package was impossible. Happily, as evidenced by the last two chapters, they were wrong. The *Notation* package allows us to work and create definitions in the language in which we are used to working. The value and utility of this accomplishment is immense.

The functions `Notation`, `Symbolize`, and `InfixNotation` were introduced in §2.2 *Notation* and §2.3 *Symbolizations, Infix Notations, and InputAliases*. These main functions were illustrated in numerous examples. In §2.4 *Options*, we described many of the options made available by the *Notation* package, such as `WorkingForm`, `Action`, and others. An introduction to box structures was given in §2.5 *Box Structures*. There the foundations were laid for the more involved discussions about notations given in the subsequent chapter. Further details about the underlying box structures, their grouping and precedences, were given in §2.6 *Precedence and Grouping of Operators*. Finally, Chapter 2 culminated in an exposition of the development and details leading up to a complete and functioning implementation of Dirac's bra-ket notation.

The present chapter continued the exploration of the foundations and working of notation statements in §3.2 *Principles behind the Notation Package*. Tag Boxes were revealed to play a pivotal role in the underlying box structures of our designed notations. Information detailing how complex patterns can be incorporated into notations was given in §3.3 *Complex Patterns in Notations*. Of great significance, in §3.4 *Tensorial Notation*, we presented a complete, functioning notation for tensors. As stated before, to the best of the author's knowledge, such a system has not been achieved by anyone previously. This is an extremely important accomplishment. Of course, some programs had *output* that was "sort of" tensor-like, but only in a strictly limited fashion. None had provided for correct input. Most importantly, one could never give new definitions in proper tensor notation, so any input was necessarily veiled, mysterious, and arcane.

Finally, in §3.5 *Tensors: Examples and Ancillary Notations*, we gave some examples and further notations involving heavy use of our tensorial notation. We also gave many definitions and rules using our tensorial objects. In short, we were using them as an integral part of the system.

In these two chapters, we have presented, detailed, and illustrated notations and their use in numerous cases. For the remainder of this thesis, notations will be used as an integral part of our working paradigm.

3.6.2 Guidelines

Before we close this chapter and indeed finish our exposition on notations, let us give some closing technical comments, advice, and guidelines for the prospective user of the *Notation* Package. The following are some such caveats and considerations to be aware of when using the *Notation* package and/or designing notations.

It is intrinsically difficult to debug something one cannot see; therefore, it is best to build up notations, seeing if something works or where a mistake has been made. It is harder to find errors if one enters a whole complex notation before testing it. Many notational problems will usually be revealed by examining the full form of an expression or its internal structure via CMD – SHIFT –e.

It is sometimes necessary to set the front end option `ShortBoxForm → False` in order to maintain notebook integrity when saving and reloading a package. This is due to the fact that the *Mathematica* parser can become confused with some forms of linear notation. This is true of both *Mathematica* 3.0 and 4.0. Although globally setting this option to `False` slows saving and loading operations, it is recommended by the author since it prevents input getting corrupted in certain circumstances.

When designing notations, one should strive to parse expressions to their correct full forms *without evaluation*. This is not always possible for complex notations, where there is no direct correspondence between an external form and an internal form. However, for the cases when it is possible, there should be no side effects from evaluation. For instance, *Mathematica* already parses expressions like $\hat{\mathcal{H}}$; thus one might then think that it would be acceptable to say

```
In[1]:= ℋ̂_ = Operator[ℋ];
```

This, unfortunately, requires evaluation to work, so is unsuitable as a proper notation. To demonstrate the absurd behavior that this approach leads to, consider the following.

```
In[2]:= Hold[ℋ ̂+ 𝒦 ] /. ℋ_ ̂+ 𝒦_ :> ℋ̂ + 𝒦̂

Out[2]= Hold[ℋ ̂+ 𝒦]
```

This result initially appears extremely perplexing. Yet, upon inspection, the problem lies with evaluation. No replacement occurs since the "operator" in the rule evaluates, whereas the "operator" inside the `Hold` does not, so in essence the calculation is the following.

```
Hold[OverHat[ℋ + 𝒦]] /. Operator[ℋ + 𝒦] :> OverHat[ℋ] + OverHat[𝒦]
```

Thus, obviously no replacement occurs. Holding, evaluation, and control of evaluation are fundamental concepts in *Mathematica*. Held expressions occur ubiquitously throughout *Mathematica*. Therefore, the approach of having notations work through evaluation is *doomed*. Moreover, declaring the formatting via `Format[Operator[ℋ_]] = ℋ̂`, leads to infinite recursion. Thus, overall, *notations should force the parsing of expressions to their full forms to occur without evaluation, and reciprocally for formatting.*

As an example of the correct approach to notations, note that the following does not depend on evaluation in order to behave correctly. (We first remove the old evaluation for *expr*.)

```
In[3]:= ℋ̂_ =.;
        Notation[ℋ̂_ ⟺ Operator[ℋ_]]

In[5]:= Hold[L̂] // FullForm
Out[5]//FullForm=
        Hold[Operator[L]]
```

The previous calculation which did not work now results in the correct behavior.

```
In[6]:= Hold[ℋ ̂+ 𝒦 ] /. ℋ_ ̂+ 𝒦_ :→ ℋ̂ + 𝒦̂

Out[6]= Hold[ℋ̂ + 𝒦̂]

In[7]:= FullForm @ %
Out[7]//FullForm=
        Hold[Plus[Operator[ℋ], Operator[𝒦]]]
```

Reciprocally, one should avoid "preemptive human evaluation" in notation statements. For instance, in contrast to the notation for Laplace transforms defined in §2.2.2 *Notation: Examples*, under no circumstances should we attempt to define that notation in the following way.

```
In[8]:= Notation[ℒ_{t_}→{s_}[f_] ⟺ ∫_0^∞ f_ e^(-s_ t_) dt_]
```

This mixes parsing and evaluation, which one should *avoid at all costs*. This ill–conceived statement is the *wrong way* to tackle the notation. We will not pursue in detail many examples illustrating why this is a fundamentally *flawed* undertaking. However, here is one.

```
In[9]:= ℒ_{t→s}[f_ + g_] := ℒ_{t→s}[f] + ℒ_{t→s}[g]

        SetDelayed::write : Tag Integrate in ∫_0^∞ e^(-s t) (f_ + g_) dt is Protected.

Out[9]= $Failed
```

As we see, we cannot make assignments for our Laplace transform objects since they parse to integrals, and thus we are trying to override the rules for `Integrate`, which of course are protected. If still curious, one can experiment to determine the numerous problems with such an approach. Briefly, the list of problem areas one would uncover include: pattern matching, pre-existing formatting rules for the right hand side, evaluation, and lack of manipulation of kernel objects. Thus always, *input needs to be parsed to kernel expressions representing the functions*. Let evaluation evaluate the functions.

The caveats above should be kept in mind by anyone designing their own notations.

3.6.3 The Future of Notations

Davenport [87], in his history of computer algebra, made the prediction that in the future, specialized computational engines will be hooked up together. An example of this would be linking, say, a polynomial solving system to a differential equation solving system. Already, the linking of systems is a widespread and familiar concept in symbolic computation. Many systems, such as *Mathematica*'s *MathLink*, Reduces's [253], MuPad's Modules, and the other dedicated communication systems such as the Multi-Protocol[133] advocated by the Open Math consortium [2, 3, 85], all tackle the same sort of problem to various degrees. However, for connecting a specialized system to a front end, the *Notation* package can play a pivotal role. For instance, among many others, systems like Magma[27, 50, 223] or Gap[121], which are largely text based specialized abstract algebra systems, are perfect candidates for using the *Notation* package as a front end for the display of their input and output. (We comment on this in §6.8.3 *GAP, Mathematica, and Tensor Simplification* in §6 *An Algorithm for Tensor Simplification*) The same is true for other specialized systems such as Schur[344].

It is an unavoidable fact that any "decent" treatment of physics in symbolic computation will involve concepts from computer science. I am concerned that some physicists, when it comes to computation, will not take the time to get the early details "right" because they just want to "get on with doing physics". Physicists are too intrenched in working out the problems they immediately face, and describing them in terms of "standard methods" and the ways that are already familiar to them. Also, they are too wedded to describing and implementing algorithms in keeping with the way they would perform the calculations on paper. Indeed, Davenport[87] affirms this general trend as well. Many programs today have such a basis, but it would be mean spirited to attack them since it is such a common place practice. Such a practice inevitably leads to closed-end designs that pay no heed to the twenty odd years of recent development of computer science. This is all the more unfortunate since computer science has witnessed a rapid development with major revolutionary steps.

It is my fervent belief that most scientific packages should be developed in order to be useful to and used by a wide community. In other words, a package should not be so complex that in the end only the developers and their direct colleagues use the software. When that is the case, I believe the authors of such software have failed, in that they have not written widely useful code. Granted, what they offer may solve some specific problem(s), and indeed they may use it

as a mill to grind out special cases or solutions to problems in order to publish. But I believe that, for the most part, this does not benefit the community as a whole.

Good notations form the bridging step from specialist systems to general use in the scientific community. Of course, that many of the authors of existing packages have not used good, let alone proper, notations has not been entirely their fault, considering the lack of software having the capabilities of the *Notation* package, at least prior to its release in late 1996. With the advent of the *Notation* package this situation has changed.

The *Notation* package is a fairly sophisticated program, and the mechanics of how the notation statements are generated will not be elaborated on in this thesis. However, it should be noted that such an undertaking is made vastly simpler by having a dynamic language in which the language can change as the program executes. *Mathematica* is such a language. In addition, the interested reader might note the extent to which the *Notation* package is written in a style that is similar to that advocated by Knuth [196], called 'Literate Programming', a label he also coined.

Wolfram Research is currently in an advanced stage of development of an extension to the *Mathematica* front end which enables the graphical editing of diagrams. This will allow diagrams to be parsable by *Mathematica*. This is not yet part of the standard *Mathematica* distribution, so a thorough discussion of it is premature at this stage. However, the graph editing structures, just like the other input structures, are made out of boxes. To parse and format such structures, one needs a program that parses and formats boxes. By the very *design* of the *Notation* package, in that it is a general program which specifically handles the translation of box structures, it can easily be extended to encompass the box structures encountered in graph editing. Indeed, the box structures it recognizes currently reside in an internal table. This has the implication that we can design notations for all sorts of diagrammatic notations in their native form. Such applications would include category theory diagrams[215, 216], Feynman diagrams[173, 186, 285], angular momentum coupling diagrams[306], etc. Barring some major unforeseen problem, when *Mathematica* releases a version containing the graph editing structures, it will also contain an updated version of the *Notation* package which can work with such structures.

As for larger concerns with notations and system design, it is mystifying to the author why unicode[316] has not yet become ubiquitous in computer algebra systems. *Mathematica* uses unicode but encodes all characters in a long ASCII form. The dependence on ASCII should have been eliminated by now. Stranger still, it appears that no other system fundamentally uses unicode either. Perhaps this is due to many programs and programmers being "text based". It was obvious to the author at the outset from *Mathematica* 3.0 Beta 1, that notations would be necessary and developers would create packages inside *Mathematica* notebooks. Within Wolfram Research, this was almost universally thought to be pure madness. With the passing of time, however, it has become evident that the original views of the author are now widespread. Many packages are developed within a notebook, and notations are widely used. Similarly, I believe the same phenomenon is true of unicode. There are no "intrinsically unicode" editors, thus unicode is not used for symbolic computation systems, thus there is no demand for "intrinsically unicode" editors, etc. However, it would seem to be the perfect solution to many problems in parsing in conjunction with symbolic computation systems. In systems that are becoming unwieldy and complex, it would be beneficial, and possibly required,

to at some stage consolidate and prune out the old concepts that are outmoded. ASCII seems to fit this description.

To summarize and strengthen our arguments about the necessity of notations, let us repeat a comparison that was evidenced previously in §3.5 *Tensors: Examples and Ancillary Notations*. In that section, two examples for the expansion of the Christoffel symbol in terms of the metric tensor were given, both using *Mathematica*, but one using notations, and the other not.

$$\Gamma^{i\_}_{k\_\,l\_} := \frac{1}{2} g^{i\,m} \left(g_{k\,m\,,\,l} + g_{m\,l\,,\,k} - g_{k\,l\,,\,m} \right)$$

```
withmetric =
   {Gam[i_, k_, l_] :>
         Module[{m},
            DefES[1 / 2 gc[i, m] * (PD[g[m, k], x[l]] +
                        PD[g[m, l], x[k]] - PD[g[k, l], x[m]]),
               {m}, ESRange -> $ESDimension]]};
```

The first is eminently understandable. The other is cryptic and inelegant at best. The first a physicist would largely recognize. The other is nearly indecipherable.

Previously, most calculations had to be conducted in a system that lacked standard notations, and hence suffered from a corresponding difficulty in coding, manipulation, and reasoning. With the advent of the *Notation* package, this situation has at the very least been drastically changed, if not solved. There is little doubt in the author's mind, that the notation package and its concepts, or something based upon its paradigm, will become the standard way notations are developed in symbolic computation systems. Indeed, the author found it strange that a limited notational system was implemented in the *Mathematica* 3.0 core in the first place.

Notations do not just allow users to have "pretty" input and output; notations demystify calculations and definitions for users and designers alike. Notations have always played a key part in mathematics, physics, and indeed the sciences. The degree to which they can demystify definitions, calculations, and indeed thinking and reasoning, can not be overstated. The *Notation* package has been conceived and developed to accomplish this aim in the context of symbolic computation. I believe, and indeed this thesis advocates, that the notation package achieves this goal.

In any development of an overall system for computation in which all the parts commingle to form a whole, a starting point must be chosen. I have chosen to start with the notations used in our computations. Now that we have attained a proficiency with these, we can progress on to changing the underlying language. This is the topic of the next chapter. Once this is complete, we can start to present physics computations in their entirety.

Chapter 4
Language Modifications

4.1 Introduction

The language of *Mathematica* is strongly based on rewrite rules. Other symbolic computation languages, and indeed other functional languages such as ML[146, 237, 259, 314], Clean[264, 265], and Haskell[22, 89, 310], use rewrite rules; but as yet none of these languages are as general and as flexible when dealing with rewrite rules as is *Mathematica*. The rewrite rule foundation of *Mathematica* extends throughout its language in a consistent and intuitive manner. The style of programming this affords is one of extreme flexibility and ease of use. However, at present there are some limitations implicit in this generalizability. Chief among them is that presently there is no general way to compile *Mathematica* code, owing to the lack of referential transparency[125] and the complex evaluation semantics. In spite of this, or perhaps because of this, there are still many refinements that can be made to the underlying *Mathematica* language.

This chapter examines several such refinements. Conceptually, these refinements encompass rule inheritance, and allow a form of object oriented-programming in a rule based system. At the implementation level, these amount to extensions to the class of rewrite rules available to the system. These extensions, although theoretically significant, have been largely made to afford ease of design in practical situations.

In this chapter we will describe how it is possible to refine *Mathematica*'s handling of assignments and transformation rules into a more general, integrated, and systematic form. The *Assign* package implements these ideas directly in *Mathematica* in a clear, intuitive, and efficient way. Once we introduce these mechanisms, they will give rise to:

- A more integrated system
- A way to write system modifying or system creating programs in an intuitive way
- A way to present assignments which are active only in certain environments
- A way to implement object-oriented programming and inheritance in rule based systems
- An inheritance paradigm that naturally incorporates the notion of multiple inheritance
- A way of tying together rule rewriting systems and inheritance.

The tying together of rule rewriting systems and inheritance gives rise to an elegant style of programming, as we will see. Parallels can be drawn to abstract data types [125, 149, 339] and uses thereof: OBJ[127, 128, 254], C++[207, 307], Eiffel[103, 235], Modula-3[142, 242], and AXIOM[177, 261]. The style also has strong parallels to the formalism of category theory, in

particular functors[263, 327]. Another field that borders on the topics presented in this chapter is generic programming[14, 176] — typified by the Haskell extension PolyP[175].

Concerning inheritance itself, in *Mathematica* we can define a versatile system that will encompass many different styles of inheritance. Yet, within the field of symbolic computation, it is important to note comparisons to the system AXIOM and to the system Gauss[243], a package for Maple[57, 244, 329]. Both AXIOM and Gauss are object-oriented, or more specifically abstract data type / category theory oriented. The system presented here is more flexible than either AXIOM or Gauss, but this comes at the price of not having as many inherent safeguards. Thus the flexibility has both strengths and weaknesses.

In §4.2 *Assignments, Rules, and Values* we start with a brief review of the differences between rules and assignments. This is followed by a review of *Mathematica*'s current system of "value" functions and some of its deficiencies. These deficiencies motivate §4.3 *Extensions of Transformation Rules*, where we introduce *tagged rules*, which simplify the handling of "values". When tagged rules are used, all values become resolvable (we can tell where they came from). This enables us, in §4.4 *Assignment and Inheritance*, to define the function `Assign` and in turn to present a simple, flexible, intuitive form of inheritance. At that point we also develop more complicated models of inheritance. These ideas are applied in §4.5 *Examples of Inheritance*, to abstract data types and stacks.

In §4.6 *Dynamic Rules and Assignments*, we extend the notion of tagged rules to that of dynamic rules. These context sensitive, non-local rules will form the backbone of much of the code that we will create in the later chapters, where we consider applications in tensor analysis, quasi-spin, general operator handling and other subjects. This chapter is rounded out with §4.7 *Technical Details* about how the *Assign* package works and §4.8 *Conclusions and Implications.*

It should also be noted that, although the concepts presented in this chapter are couched in the language of *Mathematica*, the ideas are largely applicable to rewrite rule languages in general. It may well be the case that these ideas have been presented in some abstract form in the literature at some point; but at the time of writing, there are no systems known to the author that actually have an implementation of these concepts. Indeed, the concepts presented here are almost diametrically opposed to the current prevailing trends in computer science. In computer science, highly desirable language features are referential transparency and static typing, which both facilitate compilation. (Actually certain languages like Clean[264, 265] are moving towards incorporating more dynamic code; but in the main, the trend is towards referential transparency and static typing.) Dynamic structures break referential transparency and, by their nature, are highly non-static. Yet despite this, they lead to an extremely elegant method to describe and code structures, as we will see.

It should also be pointed out that there has been work on applying the ideas of object-oriented programming to symbolic computation (and indeed to many other fields which are not so relevant to the work in this chapter). For a concrete language based on "classical" object-oriented programming see any of Eiffel[103, 235], Smalltalk[129, 234], Modula-3[142, 242], Self[301, 315], Java[167, 318], or OBJ[127, 128, 254], or to a lesser extent C++[207, 307]. For a general reference on object-oriented programming, see Budd[38] and Meyer[236]. The marrying of symbolic computation and object-oriented programming has typically occurred down "classical" lines. In general symbolic computation, one might see the work of

Temperini[91, 205, 64], Abdali & Soiffer[4], or Vlasov[324]. The general symbolic computation system AXIOM[177, 261] and, to a lesser extent, MuPAD[119] have combined symbolic computation and object-oriented programming. Some specific computer algebra systems are firmly based on object-oriented formalism, for instance Magma[27, 50, 223] and GAP[121]. In *Mathematica*, one might see the implementation by Maeder[222], or indeed for a derivative of the *Mathematica* system, see AlgBench[221, 139, 241].

Object-oriented programming has become so diverse in its paradigms and implementations — class based versus prototype based, typed versus typeless, etc. — that calling something object-oriented still leaves a huge gulf of latitude in language design. Indeed, Luca Cardelli[53] states *"...The definition of what makes a language object-oriented is still controversial. An examination of the differences between Simula, Smalltalk and other languages suggest that inheritance is the only notion critically associated with object-oriented programming. Coroutines, message-passing, static / dynamic scoping, type checking and single / multiple superclasses are all fairly independent features which may or may not be present in languages which are commonly considered object-oriented. Hence, a theory of object-oriented programming should first of all focus on the meaning of inheritance."*

It could even be argued that the language model to be presented in this chapter is not object-oriented since it is not data structure centric. Rather, in some sense, it is expression manipulation centric: the code that manipulates expressions is itself manipulated, transformed and inherited. This will all be clarified in the latter sections, after we give a careful introduction to rules as they occur in *Mathematica*.

4.2 Assignments, Rules, and Values

4.2.1 Assignments and Transformation Rules

This section is intended to give a brief outline and description of the terminology and usage of assignments and transformation rules. *Mathematica* is said to be a rule-based language. Specifically though, it uses *assignments* (or equivalently *function definitions*) and *transformation rules*. It is important to comprehend both of these concepts, which are reviewed below, and the relationship between them, covered in the next subsection, to fully understand the following sections.

Definition 4.2.A: An *assignment* is a statement of the form *lhs* = *rhs* or *lhs* := *rhs*.

An assignment (or equivalently a function definition) is just a rule that is applied, whenever possible, in the evaluation of all expressions. For example, consider the following assignment for `f`.

```
In[1]:= f[x_] := x^2
```

Any expression entered subsequently that involves f[_] will be rewritten, as we see in

```
In[2]:= g[f[Sin[x] + t]]

Out[2]= g[(t + Sin[x])^2]
```

In contrast, a *transformation* rule is applied only when the user specifies that it is to be used. For example, say we are given the transformation rule

```
In[3]:= g_Rule = g[t_] → t + c

Out[3]= g[t_] → c + t
```

Then, if we enter

```
In[4]:= g[f[Sin[x] + t]]

Out[4]= g[(t + Sin[x])^2]
```

we can see that the output expression has not been reduced/transformed by g_{Rule}. The user can, of course, decide to *apply* the transformation rule g_{Rule} as follows.

```
In[5]:= g[f[Sin[x] + t]] /. g_Rule

Out[5]= c + (t + Sin[x])^2
```

Definition 4.2.B: A *transformation rule* is a statement of the form *lhs* → *rhs* or *lhs* :→ *rhs*.

Assignments and transformation rules are fundamental to *Mathematica*, and any user of *Mathematica* should be reasonably familiar with them and able to distinguish between them. However, sometimes we will use the word 'rules' ambiguously, to refer to assignments and/or transformation rules.

4.2.2 DownValues and UpValues

What is the relationship between assignments and transformation rules? Essentially, *Mathematica* internally stores each definition (or assignment) as a corresponding transformation rule. *Mathematica* provides mechanisms for returning the transformation rules corresponding to the assignments associated with a symbol. These are the "value" functions DownValues, UpValues, OwnValues, SubValues, NValues, FormatValues, and DefaultValues. For illustration, consider the following additional assignment for f.

```
In[6]:= f /: ∂_c_ f[x_, h_] := h[x + c]
```

Now f has the following rules or assignments associated with it.

```
In[7]:= ? f

        Global`f
```

```
∂c_ f[x_, h_] ^:= h[x + c]
f[x_] := x^2
```

This is, of course, the printed form of the rules for `f`. Let us now attempt to obtain the values or transformation rules of `f`. Consider

```
In[8]:= DownValues[f]

Out[8]= {HoldPattern[f[x_]] :> x^2}
```

`DownValues[f]` returns some of the rules associated with the symbol `f` as a list of transformation rules. As one can see, there is a close resemblance between the transformation rule and the corresponding assignment. (*Mathematica* includes the wrapper `HoldPattern` in the above transformation rule to ensure that `f[x_]` itself is not evaluated to give $x\_^2$.)

However, the "downvalues" of a symbol do not necessarily give all the rules associated with that symbol. In *Mathematica*, there are sometimes rules that are associated with other "values". For example, the above assignment for $\partial_{c\_}$ `f[x_, h_]` is an "upvalue" for `f`. To obtain this assigned rule as a transformation rule, one must request the "upvalues".

```
In[9]:= UpValues[f]

Out[9]= {HoldPattern[∂c_ f[x_, h_]] :> h[x + c]}
```

This is now all the transformation rules associated with the symbol `f`. Unfortunately, in general, the `UpValues` and `DownValues` do not always comprise the full set of rules associated with a given symbol. There could have been further transformation rules returned by any of the "values" functions already mentioned: `OwnValues`, `SubValues`, `NValues`, `FormatValues`, and `DefaultValues`. Each of these is a "value" function: each returns a (possibly empty) list of transformation rules which represent assignments for the given symbol. For instance, the following assignment actually defines an "nvalue" for `foo` rather than an "upvalue".

```
In[10]:= N[foo[x_]] := numericalFunc[x]
```

Similarly, the following assignment represents a "subvalue".

```
In[11]:= foo[x_][t_] := x^2 + t
```

We can obtain these "values" by the following.

```
In[12]:= NValues[foo]
         SubValues[foo]

Out[12]= {HoldPattern[N[foo[x_]]] :> numericalFunc[x]}

Out[13]= {HoldPattern[foo[x_][t_]] :> x^2 + t}
```

However, neither of these rules would appear in the downvalues or upvalues of `foo`. Most references on *Mathematica* give more details on the different value functions. In particular, see Trott[312], Maeder[222], Wagner[325], and Wolfram[342, 343].

4.2.3 Disparate Nature of Standard Value Functions

Collectively, the examples in the previous subsection demonstrate the following fact. All the rules assigned to a symbol are *separated* as far as the user's access to them is concerned. We can diagrammatically represent this by the following figure.

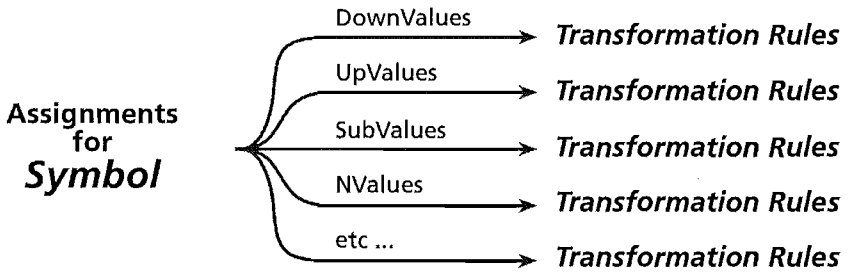


Figure 4.2.A: The disparate nature of the Value functions.

Each "value" function takes a symbol and returns a separate list of transformation rules. The most significant deficiency of this way of presenting the "values" of assignments is that the "owner" of some of the resulting transformation rules cannot in general be resolved from only the rules themselves. That is, it is sometimes impossible to tell which symbol a specific rule should be associated with if one considers just the rule. For example, consider the transformation rule

```
{HoldPattern[h[k[x_]]] :> x^3}                                    (4.2.a)
```

Is this an upvalue for `k` or a downvalue for `h`? Without knowing where the rule came from, there is no way to tell. It could be either.

It is obvious that if we are going to easily manipulate the rules or definitions associated with several symbols, then there are several things we desire:

- To be able to handle *all* such rules at once, not just the "down", "up", "N", etc... values separately.
- To be able to handle the collection of such rules programmatically.
- To be able to tell which rules are associated with which symbol.
- To have this new functionality seamlessly integrated with the rest of *Mathematica*.

We shall see that the *Assign* package enables us to accomplish these objectives. Consequently, the ideas behind, and functionality of, the package has implications for the design and structuring of information and the coding of programs in *Mathematica* and other rule based languages. Using the upcoming concepts as an underlying foundation allows many applications to be presented in a beautifully elegant way. This is true not only for computer science, but also for many other fields, for example, abstract algebra — see [147] — and for the

mathematical sciences, physics in particular — see §5 *Prototypical Structures and Quantum Mechanics* and §7 *Tensor Calculus, Applications, and Quasi-Spin*.

4.3 Extensions of Transformation Rules

4.3.1 Values

It has been demonstrated above that, using standard *Mathematica*, we can only obtain the rules of a symbol separately; however, we would like them all together. The obvious solution is just to obtain all the lists of transformation rules from the `DownValues`, from the `UpValues`, and from the rest of the value functions and join them all together. This is, in part, what `Values` does. `Values` is a function, defined in the *Assign* package, which combines all of the separate value functions of *Mathematica*.

`Values[`*f*`]`	returns a list of tagged transformation rules corresponding to all `DownValues`, `UpValues`, `OwnValues`, `SubValues`, `NValues`, `FormatValues`, `DefaultValues` and other values defined for the symbol *f*
`Values[{`*f*, *g*,...`}]`	returns all values for the symbols *f*, *g*, etc.

The function `Values`.

The values are returned as *tagged* transformation rules and *not* transformation rules. Tagged transformation rules will be defined shortly, but first, we can now view the function `Values` as follows.

Figure 4.3.A: Values.

As we will see in the next subsection, using tagged transformation rules intuitively solves our problem of not being able to determine which symbol a rule is associated with. Using these new tagged transformation rules, we can in essence join up all the values returned since we implicitly know where each rule came from. Thus we solve the problem of a disparate list of values via our new tagged rules.

4.3.2 Tagged Rules

How can we resolve which rules are associated with which symbols? For instance, as stated previously, the rule `h[k[x_]]` :→ x^3 could be a downvalue for `h` or an upvalue for `k`. In general, given a set of rules, we can not uniquely resolve the associated symbols; therefore, we must attach more information to transformation rules.

Consider the definition `f/: D[f[x_],c_]:=` *expr*. Here we are telling *Mathematica* to associate this definition with `f`, not `D`. This is done through the use of the tag `'f/:'`. In the same way, we would like to be able to include a tag with each rule, thus creating "tagged rules". For example:

```
f /: f[x_] :→ x^2
```

Each such "tagged rule" or "tagged delayed rule" must consequently have a tag symbol embedded in it. This tag symbol uniquely specifies the symbol with which the rule should be associated. In every other respect, tagged rules behave like normal rules.

data structure	*notation*	*usage*
`TaggedRule[` *lhs*, *rhs*, *tag*`]`	*tag* `/:` *lhs* → *rhs*	a transformation rule, associated with the symbol *tag*, that transforms *lhs* to *rhs*
`TaggedRuleDelayed[` *lhs*, *rhs*, *tag*`]`	*tag* `/:` *lhs* :→ *rhs*	a delayed transformation rule, associated with the symbol *tag*, that transforms *lhs* to *rhs*

The data structures for tagged rules.

To illustrate tagged rules and their properties, let us first load the *Assign* package.

```
In[1]:= <<Assign`
```

Once the package is loaded, it is permissible to enter tagged transformation rules such as the following.

```
In[2]:= g /: g[x_] → x^2

Out[2]= g[x_] → x^2
```

The *Assign* package formats this rule exactly like a standard transformation rule. This is intentional, since we would like tagged rules to behave, and appear, exactly like normal transformation rules when appropriate, and to borrow from the notation of assignments (with tags) when necessary. However, the above is still a tagged transformation rule. This is evident from its `InputForm`, as well as its `FullForm`:

```
In[3]:= FullForm @ %
```

Out[3]//FullForm=

```
TaggedRule[g[Pattern[x, Blank[]]], Power[x, 2], g]
```

There are two different kinds of "tagged" rules: `TaggedRule` and `TaggedRuleDelayed`. They directly parallel `Rule` and `RuleDelayed` (that is $\rightarrow$ and $:\!\rightarrow$). Tagged rules have the same behavior as their "untagged" counterparts except they have an embedded tag inside them.

Technical Note: To be consistent with *Mathematica*'s use of `TagSet` and `TagUnset`, it seems that one should use `TagRule` and `TagRuleDelayed` rather than our above choices. However, `TagSet` is used in the sense of "perform a `Set` operation with a tag present", hence `Tag` appearing in `TagSet` is used as a noun. In contrast, we are using 'Tagged' as an adjective modifying 'Rule', hence we prefer and will use `TaggedRule` rather than `TagRule`, and similarly for `TaggedRuleDelayed`, etc.

To further illustrate the properties of tagged rules, consider the following.

In[4]:= *rules* = {g /: y [g[x_]] → h[x], p /: p[x_, r] :↦ x^2}

Out[4]= {g /: y[g[x_]] → h[x], p[x_, r] :↦ x^2}

We can see the tag `g/:` is visible in the `OutputForm` above, but not the tag `p/:`. This is due to the fact that the head of the second rule is `p`, so we don't really need the tag to be shown *explicitly*. Tags are shown explicitly in the `OutputForm` only if the rule is not associated with the direct head of an expression.

The *Assign* package adds the option `ShowTags` to the options of `Format`. If `ShowTags` is set to `True`, then all tagged rules will appear with a tag. If `ShowTags` is set to `Automatic`, then only the strictly tagged rules will appear with tags. If `False`, then the tags will be not appear at all in the tagged rules. The Default is `Automatic`.

For the purposes of illustration, let us re-enter the rules given in the previous section for `f`.

In[5]:= f[x_] := x^2;
f /: $\partial_{c\_}$ f[x_, h_] := h[x + c];

We can now demonstrate the behavior of `Values` and of the option `ShowTags` by the following.

In[7]:= *thefValues* = Values @ f

Out[7]= {f /: f[x_] :↦ x^2, f /: $\partial_{c\_}$ f[x_, h_] :↦ h[x + c]}

`Values` has returned a list of tagged transformation rules corresponding to the assignments made for `f`.

In[8]:= ? f

```
Global`f
```

$\partial_{c\_}$ f[x_, h_] ^:= h[x + c]
f[x_] := x^2

We can change the appearance of these tagged rules by changing the option `ShowTags` of the function `Format`.

```
In[9]:= SetOptions[Format, ShowTags → True];
        thefValues

Out[10]= {f /: f[x_] :↦ x^2, f /: ∂_c_ f[x_, h_] :↦ h[x + c]}

In[11]:= SetOptions[Format, ShowTags → False];
         thefValues

Out[12]= {f[x_] :↦ x^2, ∂_c_ f[x_, h_] :↦ h[x + c]}

In[13]:= SetOptions[Format, ShowTags → Automatic];
         thefValues

Out[14]= {f /: f[x_] :↦ x^2, f /: ∂_c_ f[x_, h_] :↦ h[x + c]}
```

But in any case, the internal form of these rules remains unchanged.

Technical Note: When a rule is shown without a tag, in actuality an invisible tag box is wrapped around the output, thus it is still implicitly tagged. This is to ensure that when such a rule is re-entered, it is interpreted as a tagged rule. See §3.2.2 *Tag Boxes*.

4.3.3 Replacements and Behavior

Since tagged rules look almost the same as normal rules, intuitively they should behave like normal transformation rules. Therefore `Replace`, `ReplaceRepeated`, `ReplaceAll`, and `Dispatch` should all be able to handle tagged transformation rules in addition to standard transformation rules. Consequently, the *Assign* package modifies the behavior of these system functions slightly. Basically, if one of these system functions is called with tagged transformation rules, the tagged rules are changed to standard rules and then the system function is called with these instead. Therefore, all replacements with tagged transformation rules behave exactly like replacements with the corresponding untagged (standard) transformation rules. For example:

```
In[15]:= rules

Out[15]= {g /: y[g[x_]] → h[x], p[x_, r] :↦ x^2}

In[16]:= (y[g[Sin[t]]]) /. rules

Out[16]= h[Sin[t]]
```

In fact, the *Assign* package defines a function `ToStandardRules` that will change sets of tagged transformation rules into sets of standard transformation rules.

`ToStandardRules [` *taggedRules*`]`	return a set of standard transformation rules based upon the tagged transformation rules *taggedRules*

The function ToStandardRules.

As expected, applying `ToStandardRules` to our rule set *rules* returns a set of standard rules.

```
In[17]:= ToStandardRules @ rules
```

```
Out[17]= {y[g[x_]] → h[x], p[x_, r] :> x^2}
```

There is, however, one other important difference to note. Unlike any of the rules returned by `DownValues`, `UpValues`, `OwnValues`, `SubValues`, `NValues`, `FormatValues`, or `DefaultValues`, the tagged rules returned by `Values` do not have their left hand sides wrapped with a `HoldPattern`.

```
In[18]:= Values @ f
Out[18]= {f /: f[x_] :> x^2, f /: ∂_c_ f[x_, h_] :> h[x + c]}

In[19]:= FullForm @ %
Out[19]//FullForm=
List[TaggedRuleDelayed[f[Pattern[x, Blank[]]], Power[x, 2], f],
  TaggedRuleDelayed[D[f[Pattern[x, Blank[]], Pattern[h, Blank[]]],
    Pattern[c, Blank[]]], h[Plus[x, c]], f]]
```

However, the behavior of the tagged transformation rules returned by `Values` is only slightly different from that of normal transformation rules returned by `DownValues`, etc., since both `TaggedRule` and `TaggedRuleDelayed` hold their arguments instead of explicitly including a `HoldPattern`.

```
In[20]:= Attributes /@ {TaggedRule, TaggedRuleDelayed}
Out[20]= {{HoldAll, Protected}, {HoldAll, Protected}}
```

4.4 Assignment and Inheritance

4.4.1 Assign

Now that we have these new tagged transformation rules, what purpose do they serve? Besides cleaning up the way *Mathematica* handles "values", there is an extremely important consequence of their introduction. Figure 4.3.A for `Values` was not complete. Since every tagged rule "knows" which symbol it is associated with, we can now define a function `Assign` that is, in essence, the inverse of `Values`. Therefore, the extended diagram becomes

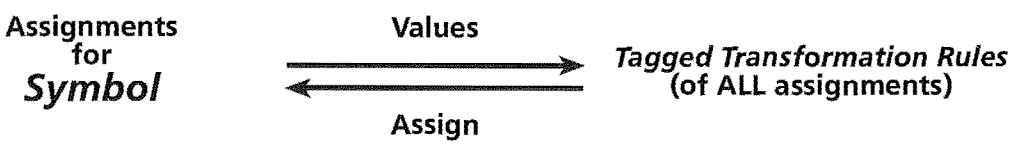


Figure 4.4.A: The relationship between Values and Assign.

Strictly, the inverse of `Values` would just give the list of symbols `Values` was called with. However, considering the semantic meaning of `Values`, we define the inverse of `Values` to be the assignment of the transformation rules to the symbols they are associated with.

data structure	*usage*
`Assign[` *rules*`]`	adds *rules* to the set of active assignments.
`UnAssign[` *rules*`]`	removes *rules* from the set of active assignments.

The functions `Assign` and `UnAssign`.

Technical Note: `Assign` can be applied to a single rule or a list of rules. Any such rule can be a standard *Mathematica* rule, a tagged rule, or a dynamic rule (as introduced in §4.6 *Dynamic Rules and Assignments*).

For example, consider assigning the tagged transformation rules in the set *rules* given below.

```
In[1]:= rules = {g /: y[g[x_]] → h[x], p /: p[x_, r] :→ x^2};

In[2]:= Assign @ rules

In[3]:= ?g

        Global`g

        y[g[x_]] ^= h[x]

In[4]:= ?p

        Global`p

        p[x_, r] := x^2
```

The above example with *rules* illustrates a most important feature of `Assign`: it allows us to perform assignments for *more* than one symbol at a time. Therefore, our diagram can be further extended as follows.

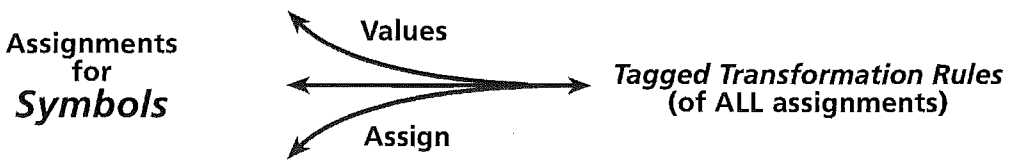


Figure 4.4.B: The relationship between Values and Assign for multiple symbols.

We close this subsection by illustrating that, just as `Assign[`*rules*`]` adds *rules* to the set of assigned rules, `UnAssign[`*rules*`]` removes *rules* from the set of assigned rules. For example, whereas above we added *rules*, the following removes *rules*.

```
In[5]:= UnAssign @ rules

In[6]:= ?g

        Global`g

In[7]:= ?p
```

```
Global`p
```

4.4.2 Inheritance

The important foundational structures have now been described. It is but a small step to achieve inheritance. Now that we have sets of tagged rules, we can change them and re-assign them. For example, consider the following set of rules for f.

```
In[8]:= ClearAll @ f;
        f[g_][t_] := g + t;
        f /: r @ f @ x_ := x^3;
        N @ f @ t_ := N @ t^3;
        Default[f, 2, 3] := 1;
        f[x_, c_] := c x^2;

In[14]:= ? f

        Global`f

        f[g_][t_] := g + t
        r[f[x_]] ^:= x^3
        f[x_, c_] := c x^2
        N[f[t_]] := N[t^3]
        f /: Default[f, 2, 3] := 1
```

We can now set

```
In[15]:= thefValues = Values @ f

Out[15]= {f /: f[x_, c_] :> c x^2, f /: r[f[x_]] :> x^3, f /: f[g_][t_] :> g + t,
           f /: N[f[t_]] :> N[t^3], f /: Default[f, 2, 3] :> 1}
```

and then assign these rules to a different symbol, say k.

```
In[16]:= Assign[thefValues /. f → k]

In[17]:= ? k

        Global`k

        k[g_][t_] := g + t
        r[k[x_]] ^:= x^3
        k[x_, c_] := c x^2
        N[k[t_]] := N[t^3]
        k /: Default[k, 2, 3] := 1
```

We can even easily remove the rules of f (or a subset of them) by

```
In[18]:= UnAssign @ thefValues

In[19]:= ? f

        Global`f
```

If we contemplate the above, it is clear that we have actually performed *inheritance* of rules. We were given some assignments on f. From these we generated a corresponding list of tagged transformation rules for f. We modified these rules by replacing f by k, that is, *rules*/. f → k. Then we assigned the resulting tagged transformation rules. This process results in k having the same rules and assignments as f. In short, k has inherited the rules of f; or equivalently, the rules of f have been inherited to k.

It is now evident that inheritance can be very elegantly performed by the sequence

Assign @ *Replacements* @ Values @ *Symbols* (4.4.a)

This inheritance mechanism can be diagrammatically represented as follows.

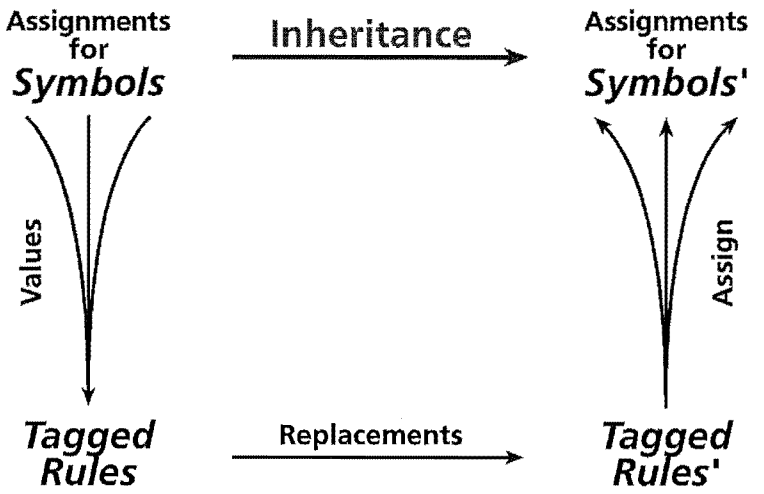


Figure 4.4.C: The process of inheritance through tagged rules.

Here is a simple example illustrating inheritance of several symbols, occurring simultaneously.

```
In[20]:= rules
Out[20]= {g /: y[g[x_]] -> h[x], p[x_, r] :> x^2}
In[21]:= Assign[rules /. {y -> foo, g -> w, r -> expr}]
```

Consequently, the symbols w and p are bestowed with the following rules.

```
In[22]:= ?w
         Global`w
         foo[w[x_]] ^= h[x]
In[23]:= ?p
         Global`p
         p[x_, expr] := x^2
```

Note that the previous rules for p, from past assignments, remain unaffected.

In this whole paradigm, multiple inheritance is totally natural. For instance, consider the following definitions.

```
In[24]:= h[a_] := a^2
         r[t_, y_] := t + y
```

We can now arrange to have a single symbol, say goo, inherit both of these definitions.

```
In[26]:= Assign[Values[h] /. h → goo];
         Assign[Values[r] /. r → goo];

In[28]:= ? goo

         Global`goo

         goo[a_] := a^2
         goo[t_, y_] := t + y
```

We have described the properties and behavior of inheritance in terms of some g inheriting the properties of some f. Many readers might be wondering how the above theory is applied. The next section gives a quick example that is somewhat contrived. Yet, the bulk of the applications are given in §5 *Prototypical Structures and Quantum Mechanics* and §7 *Tensor Calculus, Applications, and Quasi-Spin*, once further, and arguably necessary, additions are given in §4.6 *Dynamic Rules and Assignments*. There we will see another extension that will finally allow us beautifully simple specifications of the structures in which we wish to perform computations.

4.5 Examples of Inheritance

4.5.1 Stacks via Abstract Data Types

To illustrate a very simple example of the model of inheritance introduced in the previous section, we can examine the modeling of a generic stack and the associated inherited stacks. Since the concept of a stack is extremely widely known in computer science [89, 125, etc.], stacks are one of a handful of examples that are usually treated in language definitions and explanations. Thus, in part to conform to convention and also because they are so simple, in this chapter we choose to illustrate inheritance using stacks. The presentation in this section mirrors the presentation of abstract data types as typically used in computer science [89, 125, 149, 339]. For those readers not familiar with such concepts, the presentation is meant to give a basic impression of abstract data types and of their workings.

The use of abstract data types provides a conceptual framework for modelling generic or abstract properties of a system[125, 149, 339]. Abstract data types are so named because they are abstracted from any implementation layer. They specify how the types associated with a structure should behave rather than focusing on implementing the functioning of a structure.

For instance, we know that the "length" of a list should be an integer, but this tells us nothing about how to implement the function 'length'. *Mathematica* is not a strongly typed language, but we can, of course, emulate types in *Mathematica* in many and varied ways.

Technical Note: Typing information is often useful for both program correctness and compilation [6]. Even in the case of general computer science, many varied and difficult questions involving typing arise -- see for instance Cardelli[53]. However, in the area of symbolic computation, the problems associated with typing are far more complex. Some languages such as AXIOM[177] and REDLOG[95] try, and arguably succeed to some degree, to have typing information inherently tied to the underlying programming language. However even for these languages, problems can, and do, arise. In a "natural" typing system for symbolic computation, types must be generalized to some degree. For instance, a system might type the number 3 as a positive integer. However, if we make the assumption about x that it is an integer lying between -2 and 2, then what is its type? Such questions can not be directly answered using many of the current typing models. Herein lies the difficulty of using strong typing. Instead, we have to view typing information as generalized predicates or sets of equations and inequations. Unfortunately, the "genericity" of the typing structures currently being researched and implemented are still not sufficiently sophisticated to handle nicely the problem of typing in symbolic computation.

The generic stack structure we are about to implement can be approached in two ways. One way is functional and referentially transparent; the other is more imperative in nature and includes side effects. Hence, we will use two different naming schemes: $name_f$ for functional and $name_s$ for side effects. Let us symbolize these names so that we can use them in functions.

```
In[1]:= Symbolize[expr_f]; Symbolize[expr_s]
```

Let us also create a nice notation for `HoldPattern`, since we will use it so frequently.

```
In[2]:= Notation[expr_hp ⟺ HoldPattern[expr_]]
```

Technical Note: Actually, this notation is one of those given in §A.1 *Common Notations*, but we have created it in isolation since it is the only one needed for this section.

We now proceed to give the rules for a typed generic stack. As with all *Mathematica* functions / programs, there are many ways in which we could implement such a concept using rules. Here is one such way. (We use '`StackT`' in place of '`Stack`', as the latter already has a meaning in normal *Mathematica*. The Gothic letter '`T`' is meant to be suggestive of a "type" or of a prototypical/generic type structure.)

```
In[3]:= StackRules_f = {
          StackQ[StackT @ elms___?StackElementQ]_hp → True,
          StackQ[other___]_hp → False,

          EmptyQ[StackT[]]_hp → True,
          EmptyQ[_StackT?StackQ]_hp → False,
          EmptyQ[other___]_hp → $Failed,

          Top_f[stack_StackT]_hp :↦ First @ stack /; ¬ EmptyQ @ stack,
          Pop_f[stack_StackT]_hp :↦ Rest @ stack /; ¬ EmptyQ @ stack,
          Push_f[elm_?StackElementQ, stack_StackT]_hp :↦
            Join[StackT @ elm, stack]};
```

This generic prototype contains the typing information, together with the underlying implementation information. The typing is emulated by the pattern `_StackT` and the testing

predicates ?StackElementQ,and ?StackQ. The inheritance mechanism described in §4.4.2 *Inheritance* is now, of course, easily applied. We can implement a stack of integers simply by:

```
In[4]:= Assign[StackRules_f /. {StackT → IntegerStack,
                StackQ → IntegerStackQ, StackElementQ → IntegerQ}]
```

Here are some simple calculations with a stack of integers.

```
In[5]:= stack = Push_f[23, IntegerStack[12, 34, 56]]

Out[5]= IntegerStack[23, 12, 34, 56]

In[6]:= Pop_f @ %

Out[6]= IntegerStack[12, 34, 56]

In[7]:= Top_f @ %

Out[7]= 12

In[8]:= IntegerStackQ @ stack

Out[8]= True
```

By the choice of replacements and by the naming choices, we have made $\texttt{Top}_f$, $\texttt{Pop}_f$, and $\texttt{Push}_f$ *polymorphic* over different kinds of stacks. The stack structure is inherited from $\texttt{StackRules}_f$. This inheritance paradigm is similar to prototypical inheritance — see for example Blaschek[24], Self[315, 301], Omega[24], and other general references on object oriented programming [18, 38, 236].

Technical Note: In practice, the above polymorphism might be called *ad-hock polymorphism* [18, 89, 264]. This is the reverse of the common situation in abstract data type languages, where it is in general nicer to define functions that are parametrically polymorphic — see for instance Haskell [22, 89, 310], Clean[264], and ML[146, 237, 259, 314]. However, *Mathematica*, by default, is in essence inherently parametrically polymorphic due to its pattern matching capabilities.

For a further simple example, we can implement stacks of atomic objects simply by the following.

```
In[9]:= Assign[StackRules_f /. {StackT → AtomicStack,
                StackQ → AtomicStackQ, StackElementQ → AtomQ}]

In[10]:= Push_f[x, AtomicStack[bob, 23, "sid", t]]

Out[10]= AtomicStack[x, bob, 23, sid, t]

In[11]:= Pop_f @ %

Out[11]= AtomicStack[bob, 23, sid, t]

In[12]:= Top_f @ %

Out[12]= bob
```

4.5.2 Type Coherence

In the examples of the previous subsection we have not, by intention, automatically generated a `$Failed` when we do not get a type match. For instance, as desired, we cannot push a composite object onto a stack of atomic objects.

```
In[13]:= Push_f[f[x], AtomicStack[12, 34, 56]]

Out[13]= Push_f[f[x], AtomicStack[12, 34, 56]]
```

Instead of failing, evaluation just returns the original expression, in keeping with the default practices of standard *Mathematica*. This subsection details the simple steps necessary to extend our original set of prototypical rules to enforce type coherence, or to modify our original set of rules for other purposes.

Let us create a new set of prototypical rules which, once instantiated and assigned, will modify our stack functions in such a way as to ensure strict type coherence.

```
In[14]:= StackStrictTypingRules_f = {
            Top_f[_StackT]_hp → $Failed,
            Pop_f[_StackT]_hp → $Failed,
            Push_f[elm_, _StackT]_hp → $Failed};
```

Instead of an overarching approach, we can selectively make a given stack class strictly type coherent in its arguments by inheriting the rules of `StackStrictTypingRules`. We illustrate this for atomic stacks.

```
In[15]:= Assign[StackStrictTypingRules_f /. {
                StackT → AtomicStack,
                StackQ → AtomicStackQ,
                StackElementQ → AtomQ}]
```

Now, when we try to push non-atomic objects onto an atomic stack, it will result in an error.

```
In[16]:= Push_f[f[x], AtomicStack[12, 34, 56]]

Out[16]= $Failed
```

Technical Note: Obviously, we could have created a specific error message to indicate, say, that we were trying to push the object '1' on a stack of '2', etc. But the idea is clear from the above, so it is left to the reader to perform any such extensions.

Alternatively, we could create a set of rules for strictly typed stacks by combining the set `StackRules` with the set `StackStrictTypingRules`.

```
In[17]:= StrictStackRules_f = Join[StackRules_f, StackStrictTypingRules_f];
```

Then, the strict stack prototypical rules, $\texttt{StrictStackRules}_f$, can be used whenever we need a strictly typed stack. On a slightly different bent, if we want, say, a typeless stack, we can just

strip off all occurrences of the typing predicate StackElementQ present in $\texttt{StackRules}_f$, using a fairly simple replacement.

```
In[18]:= Assign[StackRules_f //. {
              StackT → TypelessStack,
              PatternTest_hp [pat_, StackElementQ] → pat}]
```

We can check that typeless stacks, including their functions Push, Top, etc., place no restrictions on their members.

```
In[19]:= ? Push_f

         Global`Push__Subscript__f

         Push_f [elm_?IntegerQ, stack_IntegerStack] := Join[IntegerStack[elm], stack]
         Push_f [elm_?AtomQ, stack_AtomicStack] := Join[AtomicStack[elm], stack]
         Push_f [elm_, _AtomicStack] = $Failed
         Push_f [elm_, stack_TypelessStack] := Join[TypelessStack[elm], stack]

In[20]:= ? Top_f

         Global`Top__Subscript__f

         Top_f [stack_IntegerStack] := First[stack] /; ! (EmptyQ[stack])
         Top_f [stack_AtomicStack] := First[stack] /; ! (EmptyQ[stack])
         Top_f [_AtomicStack] = $Failed
         Top_f [stack_TypelessStack] := First[stack] /; ! (EmptyQ[stack])
```

Here is a simple example involving a typeless stack.

```
In[21]:= Push_f [f[x], TypelessStack[asym, "bye", 3]]

Out[21]= TypelessStack[f[x], asym, bye, 3]

In[22]:= Pop_f [%]

Out[22]= TypelessStack[asym, bye, 3]

In[23]:= Top_f [%]

Out[23]= asym
```

To summarize our overall example, we have considered a simple prototypical set of rules for a stack (or a specification, of sorts, for a stack), which we have inherited to several specific stack types, namely IntegerStack, AtomicStack and TypelessStack. Furthermore, we extended the specification of the stack to enforce strict type coherence and selectively inherited this for the AtomicStack.

4.5.3 Imperative Stacks

Readers with a background more in keeping with imperative languages (procedural languages), rather than functional languages, might have expected a slightly different treatment. They might, in fact, have expected usages more like the following.

```
s = IntegerStack[12, 34, 56];
Push[52, s];
Top[s];
```

However, implicit in such an approach is the use of variables with state. `Pop[s]` changes the state of `s`, so executing `Pop[s]` again has a different effect the second time around. Consequently, referential transparency is lost [125]. The previous code was written from a more functional bent, hence avoided side effects. But *Mathematica* is quite flexible and has no particular difficulty in handling state. Let us assume that the reader wants to use stacks in a more conventional or "imperative", hence non-functional, way. Then it is appropriate that we have "imperative" stacks inherit their rules from those so far developed for functional stacks, and add some new ones.

```
In[24]:= StackRules_s =
           Join[StackRules_f /. {Top_f → Top_s, Pop_f → Pop_s, Push_f → Push_s},
             {
               Top_s[stack_Symbol?StackQ]_hp :↦ First[stack] /; ¬ EmptyQ[stack],
               Pop_s[stack_Symbol?StackQ]_hp :↦ (stack = Rest[stack]),
               Push_s[elm_?StackElementQ, stack_Symbol?StackQ]_hp :↦
                 (stack = Prepend[stack, elm]),
               Attributes[Top_s]_hp → {HoldAll},
               Attributes[Pop_s]_hp → {HoldAll},
               Attributes[Push_s]_hp → {HoldAll}}];
```

Our new functions, which include state, have to hold all of their arguments, so that may change the value of the symbol referencing the stack.

```
In[25]:= Assign[StackRules_s /. {StackT → StringStack,
                 StackQ → StringStackQ, StackElementQ → StringQ}]

In[26]:= s = StringStack["was", "here"];

In[27]:= Push_s["Heisenberg", s];
         Push_s["Maybe", s];

In[29]:= s

Out[29]= StringStack[Maybe, Heisenberg, was, here]

In[30]:= Top_s[s]

Out[30]= Maybe
```

4.5.4 Tagged Stacks: A Refinement

The forgoing subsections form a perfect backdrop for an example showing the superiority of tagged rules over non-tagged rules. In particular, in §4.5.1 *Stacks via Abstract Data Types* and §4.5.2 *Type Coherence*, we created a set of prototypical rules for `Top`, `Pop`, and `Push` for a generic stack. We then inherited these rules to several concrete stack types, for example, to `IntegerStack`. Let us now consider the manipulation of the set of rules involving, say, `IntegerStack`. To be precise, what are the rules of `IntegerStack`? We would like to be able to enter `Values @ IntegerStack` and get back all the rules to do with integer stacks. However, as one can easily check, no rules have so far been explicitly assigned to the symbol `IntegerStack`.

```
In[31]:= ?IntegerStack

         Global`IntegerStack
```

Instead, all the rules involving `IntegerStack` were actually assigned to other symbols, namely `IntegerStackQ`, `EmptyQ`, $\texttt{Top}_f$, $\texttt{Pop}_f$, and $\texttt{Push}_f$. We could, of course, obtain all of the values of these last five symbols and from these select only those rules that involve `IntegerStack`; but this would be quite ugly.

The resolution of this quandary is to have the transformation rules tagged from the very beginning. In this way, we can access just the rules we want. To illustrate this, let us remove the rules so far assigned which involve `IntegerStack`.

```
In[32]:= ClearAll [Top_f, Pop_f, Push_f, IntegerStackQ, EmptyQ];
```

Let us, instead, now use the following set of tagged prototypical rules.

```
In[33]:= TaggedStackRules_f = {
           StackT /: StackQ [ StackT @ elms___?StackElementQ]_hp → True,
           StackQ [ other___]_hp → False,

           StackT /: EmptyQ [ StackT[]]_hp → True,
           StackT /: EmptyQ [ _StackT?StackQ]_hp → False,
           EmptyQ [ other___]_hp → $Failed,

           StackT /: Top_f [ stack_StackT]_hp :↦ First @ stack /; ¬ EmptyQ @ stack,
           StackT /: Pop_f [ stack_StackT]_hp :↦ Rest @ stack /; ¬ EmptyQ @ stack,
           StackT /: Push_f [elm_?StackElementQ, stack_StackT]_hp :↦
              Join[StackT @ elm, stack] };
```

Technical Note: Actually, we could have obtained the above set of rules with a tricky set of replacements on the old set of rules, $\texttt{StackRules}_f$, but it would have clouded the discussion.

Now, as before, let us inherit the generic tagged stack rules to an integer stack.

```
In[34]:= Assign[TaggedStackRules$_f$ /. {StackT → IntegerStack,
                StackQ → IntegerStackQ, StackElementQ → IntegerQ}];

In[35]:= Push$_f$[15, IntegerStack[2, 37, -9, 6]]

Out[35]= IntegerStack[15, 2, 37, -9, 6]
```

However, since all the rules were tagged, we can now determine where they came from.

```
In[36]:= Values @ IntegerStack

Out[36]= {IntegerStack /: EmptyQ[IntegerStack[]]$_{hp}$ :↦ True, IntegerStack /:
            IntegerStackQ[IntegerStack[elms___?IntegerQ]]$_{hp}$ :↦ True,
          IntegerStack /: EmptyQ[_IntegerStack?IntegerStackQ]$_{hp}$ :↦ False,
          IntegerStack /: Top$_f$[stack_IntegerStack]$_{hp}$ :↦
            First[stack] /; ! (EmptyQ[stack]), IntegerStack /:
            Pop$_f$[stack_IntegerStack]$_{hp}$ :↦ Rest[stack] /; ! (EmptyQ[stack]),
          IntegerStack /: Push$_f$[elm_?IntegerQ, stack_IntegerStack]$_{hp}$ :↦
            Join[IntegerStack[elm], stack]}

In[37]:= UnAssign @ %

In[38]:= ?IntegerStack

         Global`IntegerStack
```

To close this subsection, notice that the left hand side of almost every rule in our example rule sets has been wrapped by a `HoldPattern`. This gives a strong indication of the important role of `HoldPattern` in rules, as it pertains to our inheritance model. To elaborate, consider the following seemingly innocuous rule.

```
In[39]:= myRule = {myConstQ @ other_ :↦ False};
```

Now assume that at some later stage the rule is assigned, like so.

```
In[40]:= Assign @ myRule
```

Then, owing to the assignment of the rule for `myConstQ`, the rule set `myRule` will have been changed.

```
In[41]:= myRule

Out[41]= {False :↦ False}
```

Thus, we could not subsequently use this rule set, since it would obviously lead to erroneous results. However, if we had included a `HoldPattern` around the left hand side of the rule, then this problem would have been avoided.

```
In[42]:= myRule = {myConstQ[ other_ ]$_{hp}$ :↦ False};
```

Consequently, wrapping the left hand sides of a rule with a `HoldPattern` is usually the preferable option. It should be pointed out that some automatic stripping of superfluous occurrences of `HoldPattern` is committed by the `Values` function, but this stripping only occurs when the evaluation status remains essentially unchanged.

In these past subsections, we have seen just the beginnings of a system for how to structure rules and manipulate rule sets, allowing rule sets to flux into and flux out of assignment, *under the control of the user*. This flexibility in rule set usage is what underlies the assignment and inheritance package. We can now use this base to dramatic effect with dynamic rules, which we introduce in the next section. This generalization will allow sets of dynamic rules to flux into and flux out of assignment, *under the control of the program.*

4.6 Dynamic Rules and Assignments

4.6.1 Motivation for Dynamic Rules

The inheritance paradigm introduced in §4.4 *Assignment and Inheritance* allows us to make an elegant generalization to rewrite rules and tagged rewrite rules. This section introduces the concept of a *dynamic rewrite rule*. Again, for motivation we do not use a "real world" example since the complexity would only cloud the underlying issues. However, the basic subject of our example occurs frequently in practice: how to handle expansions which are desired only in special circumstances.

For our "toy" example, say we would like expressions of the specialized form `L[a+b]` to "expand" to `L[a]+L[b]`. Assume we would also like to avoid the use of a special expand function and just use `Expand`. A naïve solution would be the following.

```
Expand[L[a_ + b_]] := L[a] + L[b]                    (4.6.a)
```

As stated, this "solution" is highly inadequate. Yet, in many real situations, it occurs that we need to have available special expansions. For example, tensor expand, derivative expand, sum expand, group product expand, etc. In most of these cases, the expressions being expanded involve specialized data structures. This is an *extremely* common sort of situation, and can be found in many *Mathematica* programs that do "mathematical calculations" (as opposed to utility programs, such as the *Notation* package).

Let us expose the specific flaws with the above approach, (4.6.a). First, as a minor preliminary, `Expand` is a protected function, so (4.6.a) or any more sophisticated version of it, and any other new rules needed for `Expand` would have to be entered in a sequence of the form `Unprotect[Expand], ..., Protect[Expand]`.

Next, Expand needs to be "threaded" over its arguments. For instance, the above rule would not work on c L[u+v]. We would need to introduce a further rule, like the following, to rectify this.

```
Expand[x_ y_] := Expand[x] Expand[y]                    (4.6.b)
```

We could even introduce further rules so that expressions like foo[c L[u+v]] were expanded properly. However, care would have to be taken over the evaluation properties of the various "heads" in the expression. For instance, if L[u+v] were buried in an expression whose head had the attribute HoldAll, and this expression was in turn in an Expand environment, then no expansion of L[u+v] should take place.

In addition to all of this, in the original rule (4.6.a), *a* and *b* might need further expansion, so (4.6.a) should really be amended to at least the following.

```
Expand[L[a_ + b_]] := Expand[L[a]] + Expand[L[b]]       (4.6.c)
```

However, the combined result above might still need more expansion, so we need something like:

```
Expand[L[a_ + b_]] :=
    FixedPoint[Expand[L[Expand[a]] + L[Expand[b]]]]     (4.6.d)
```

Regrettably, this whole issue was spawned from the fact that we would like "L-expressions" expanded as L[a+b]→L[a]+L[b]. Typically, most designers, being cognizant of the above problems associated with the assignment approach, would instead turn to rewrite rules. Using rewrite rules as a solution is adequate but not entirely satisfactory. It might proceed along the following lines.

```
Expand[any_] := any //. {L[a_ + b_] → L[a] + L[b]}
```

However, if we need to add a second expansion rule, we must totally redefine the function. Again, typically, the way to circumvent this is to introduce a named set of transformation rules, like so.

```
ℰxpandRules = {L[a + b] → L[a] + L[b]};
Expand[any_] := any //. ℰxpandRules
```

New expansions can now be added by simply appending new rules to *ℰxpandRules*, as in AppendTo[*ℰxpandRules*, *newRule*]. Unfortunately, there is the further problem that the original argument in the above is not expanded. One cannot simply rectify this by something akin to the following.

```
Expand[any_] := Expand[any //. ℰxpandRules]             (4.6.e)
```

since the use of (4.6.e) would lead to an infinite loop. We could get around this infinite recursion by using some trickery to make sure the rule is applied only once. But even then, the expansion rule is not truly consistent with the infinite evaluation model of *Mathematica*. (A nice treatment of rule sets is given in Maeder[222].)

Thus, almost always in practice, instead of adding new rules to *Mathematica*'s `Expand`, programmers will add specific functions for *each* kind of expansion. Fortuitously, it is sometimes necessary, due to speed considerations, to have these individual expansion functions. However, it can be handy to also have just plain expansion and have the various data structures do their appropriate things according to how they should be expanded. In our particular example, we would like basically to say that `L[a_ + b_] := L[a] + L[b]` only under expansion. Using *dynamic assignments and dynamic rewrite rules*, introduced next, we will be able to do just that.

4.6.2 Dynamic Rules and Assignments I

A *dynamic rewrite rule* is a *non-local context specific* rule with a tag. This subsection describes and explains dynamic rewrite rules and dynamic assignments, and gives some simple examples of how they are used.

The following gives a table of the dynamic data structures, both dynamic assignments and dynamic transformation rules. Paralleling our sometimes loose use of "rules" and "tagged rules", we will sometimes refer to both types of dynamic data structures as "dynamic rules", when convenient and no confusion is possible.

data structure	*notation*	*usage*
`DynamicRule[` *lhs*, *rhs*, *tag*, *env*`]`	*env* ↗ *tag* / : *lhs* → *rhs*	a transformation rule, *lhs* → *rhs*, associated with the symbol *tag* and only active when used on an expression that is in an environment *env*
`DynamicRuleDelayed[` *lhs*, *rhs*, *tag*, *env*`]`	*env* ↗ *tag* / : *lhs* :→ *rhs*	a delayed transformation rule, *lhs* :→ *rhs*, associated with the symbol *tag* and only active when used on an expression that is in an environment *env*
`DynamicSet[` *lhs*, *rhs*, *tag*, *env*`]`	*env* ↗ *tag* / : *lhs* = *rhs*	an immediate assignment, *lhs* = *rhs*, associated with the symbol *tag* and only active in the environment *env*
`DynamicSetDelayed[` *lhs*, *rhs*, *tag*, *env*`]`	*env* ↗ *tag* / : *lhs* := *rhs*	an delayed assignment, *lhs* := *rhs*, associated with the symbol *tag* and only active in the environment *env*

The data structures for dynamic rules.

Technical Note: We could have used '⊢' instead of '↗' since '⊢' has a meaning in logic that has similarities to the operation we are trying to encapsulate — see any standard logic text, for example, Enderton[104].

The most unusual and important new feature of these rules is that they have a "context" or, as we will say, an *environment* under which they are active.

Technical Note: Instead of using the term 'environment', we could instead have used the term 'context'; but the latter term already has a predefined meaning in *Mathematica*, so the terminology of environments is preferable.

Definition 4.6.A: An expression *expr* occurs in the *environment*, say *env*, if *expr* is a subexpression of some expression with head *env*. The head *env* must be an acceptable *Mathematica* symbol.

For example, consider

```
Expand[x + L[a + b]] + L[a + b]
```

Here, the first `L[a+b]` occurs in the environment of `Expand`, but the second `L[a+b]` does not. For any given dynamic rule, all subexpressions within the "environment" head can potentially be affected, whereas expressions outside the environment cannot be affected. To illustrate, consider the following dynamic rule.

```
In[1]:= foo ↗ L /: L[a_ + b_] :↦ L[a] + L[b]
Out[1]= foo ↗ L[a_ + b_] :↦ L[a] + L[b]
```

Here is its full form.

```
In[2]:= FullForm @ %
Out[2]//FullForm=
DynamicRuleDelayed[L[Plus[Pattern[a, Blank[]], Pattern[b, Blank[]]]],
   Plus[L[a], L[b]], L, foo]
```

This is a dynamic rule for an expression matching `L[a_+b_]`, associated with the tag `L`, and is applicable only within subexpressions of an expression with head `foo`. A dynamic rule, by itself, is inert, as is a normal rule. We can perform replacements with this rule in a similar manner to normal replacements, although replacements can only occur within the environment `foo`.

```
In[3]:= foo[x + L[a + b^2]] + L[i + j] /. %
Out[3]= foo[x + L[a] + L[b^2]] + L[i + j]
```

It is clear that our dynamic rule only affects subexpressions inside the `foo` environment.

Technical Note: For our illustration, we used the head `foo` rather than `Expand` (as was used in the motivational example in the previous subsection). `Expand` was not used since it is normally active and hence would have expanded its argument before the replacement could have been carried out. We could easily have wrapped our expression in suitable holding functions, but to keep our illustrations simple, we have not done so.

4.6.3 Dynamic Rules and Assignments II

For each dynamic transformation, there is a corresponding dynamic assignment, just as to each standard transformation, there is a corresponding standard assignment. For instance, for the dynamic transformation rule `Expand ↗ L/:L[a_+b_] :↦ L[a]+L[b]`, the corresponding dynamic assignment is

```
In[4]:= Expand ↗ L /: L[a_ + b_] := L[a] + L[b]
```

Now the expansion of our additively linear function will occur via this dynamic assignment.

```
In[5]:= Expand[x + L[a + b^2 + c^3]] + L[i + j]

Out[5]= x + L[a] + L[b^2] + L[c^3] + L[i + j]
```

Expand now also works on expressions involving L at any depth.

```
In[6]:= Expand [ L[a + b] L[c + d] - Sin @ f @ e^L[u+v] ]

Out[6]= L[a] L[c] + L[b] L[c] + L[a] L[d] + L[b] L[d] - Sin[f[e^(L[u]+L[v])]]
```

Expand also works on chains of expressions involving L and respects the evaluation properties of the various "heads" of expressions within an Expand environment, as shown next.

```
In[7]:= SetAttributes[hoo, HoldAll]
        Expand[L[a + L[b + L[c + d]]] + hoo[L[u + v]]]

Out[8]= hoo[L[u + v]] + L[a] + L[L[b]] + L[L[L[c]]] + L[L[L[d]]]
```

In order to illustrate the use of a tag differing from the actual head, as well as illustrating the dynamic overriding of a protected symbol, consider the following somewhat contrived example.

```
In[9]:= Expand ↗ e /: e^(a_+b_)^n_ := e^Expand[(a+b)^n]

        DynamicValues::ProtectionWarning :
          The symbol e is protected but will be dynamically overridden.
```

(The warning tells us that e is normally protected but that it will be overridden dynamically.) Now the expansion of exponentials will occur via this dynamic rule assignment.

```
In[10]:= Expand[7 e^(u+v+1)^2] + e^(i+j)^2

Out[10]= e^(i+j)^2 + 7 e^(1+2 u+u^2+2 v+2 u v+v^2
```

Actually, to avoid cluttered output, let us turn off the warning that certain functions will be dynamically overridden even though they are normally protected.

```
In[11]:= Off[DynamicValues::ProtectionWarning];
```

As an extremely brief precursor to the full details in §4.7.5 *Underpinnings of Dynamic Assignments*, the following is the essence of how the dynamic assignment is working in the above situation. Assume we are evaluating an expression of the form `Expand[expr]`. At an intuitive level the following steps then take place.

```
L[a_ + b_] := L[a] + L[b];
answer = Expand @ expr;
L[a_ + b_] =.;
answer
```

Thus, intuitively speaking, once one is inside an Expand environment, one can think of the assignment `L[a_ + b_] := L[a] + L[b]` as being unconditionally present and being used together with *Mathematica*'s full infinite evaluation model to evaluate *expr*. So of course all the difficulties raised in the motivation subsection disappear.

Finally, and in summary, the above examples of dynamic assignment show that we can now actually achieve what was asked for at the end of the previous motivational subsection: we can now define rules to be active within specific environments.

4.6.4 Dynamic Rules and Assignments III

Before we proceed with some extended examples of dynamic assignments, let us first mention a few details about dynamic objects. Although all of the various dynamic rules and assignments must have a tag associated with them, the tag can be omitted in the input notation when it can be correctly determined from the left hand side of the rule or assignment. For instance, consider the following dynamic rule.

```
In[12]:= foo ↗ f[x_] → x^3
Out[12]= foo ↗ f[x_] → x^3
```

In both input and output, this rule appears not to have a tag associated with it. However, examining its full form shows that it does indeed possess the tag `f`.

```
In[13]:= FullForm @ %
Out[13]//FullForm=
        DynamicRule[f[Pattern[x, Blank[]]], Power[x, 3], f, foo]
```

This is exactly analogous to whether the tags in the output of tagged rules are visible, as discussed in §4.3.2 *Tagged Rules*. In fact, the `Format` option `ShowTags`, which controlled the visibility of tags in the output of tagged rules, has exactly the same behavior for the output of dynamic rules.

Given the similarities between dynamic rules and tagged rules, it should be no surprise that, by design, `Assign` can assign dynamic rules. Let us illustrate this by assigning the previous dynamic rule.

```
In[14]:= Assign @ %
```

Now `f` has a specialized behavior inside the `foo` environment.

```
In[15]:= foo[f[t] + x] + f[y]
Out[15]= f[y] + foo[t^3 + x]
```

Indeed, the function `Values`, by design, respects these dynamic rules and will include them in the returned list of values.

```
In[16]:= Values @ f
Out[16]= {foo ↗ f[x_] :↦ x^3}
```

Information about dynamic assignments is also contained in the information for symbols. For instance, `foo` has some environment rules associated with it, as well as `f` having some dynamic values associated with it.

```
In[17]:= ?? foo

         Global`foo

         foo has the following environment rules :

         foo ↗ f[x_] := x^3

In[18]:= ? f

         Global`f

         foo ↗ f[x_] := x^3
```

Since `Values` incorporates dynamic assignments by returning dynamic rules, we can consequently include dynamic rules in the inheritance paradigm introduced in §4.4.1 *Assign*. Thus, we can create generic rule sets specifying not only how a structure acts, but also how other structures interact with it. We will see some "real world" examples of such inheritance later in §5 *Prototypical Structures and Quantum Mechanics* and §7 *Tensor Calculus, Applications, and Quasi-Spin*.

Technical Note: Dynamic rules and assignments should not be confused with dynamic evaluation as used in certain fields of computer science. In these fields, dynamic evaluation can refer to returning multiple answers to a single question. For an example of this type of evaluation in AXIOM, see Broadbery[32], or more generally, see Duval[100, 101]. Cursorily, dynamic evaluation occurs when, in the intermediate stages of a calculation, the algorithm arrives at a critical stage where the answer could be one of several possibilities. Each of the various possibilities is investigated and the answer returned contains multiple solutions. Dynamic evaluation, as the afore mentioned authors use it, attempts to allow the reuse of all computations up until the point where the process must be split or forked. For a typical instance where multiple solutions are necessary, observe that `Reduce[a x^2 + 2 x + 1 == 0, x]` yields solutions which include the case when `a==0`; however, if we issue the command `Solve[a x^2 + 2 x + 1 == 0, x]`, then *Mathematica* would automatically assume that `a` is non-zero. This is a well-known caveat of mainstream symbolic computation systems. But in all fairness, if a system did keep track of all possible values and conditions, it could lead to intermediate constraint swell. For instance, this arises in the verification of the direct reduction method in §6.14.7 *The Method of Direct Reduction*, which is used in the `Canonicalize` algorithm of §6 *An Algorithm for Tensor Simplification*. For a system that automatically keeps track of all possible variable restrictions, see REDLOG[95]. For an extension to *Mathematica* that incorporates semantic matching and the returning of dynamic rules see Harris[148].

To recap, standard assignments are globally active, but a dynamic assignment is active only in a specific environment. For instance, we might want the rule ABC to be active under simplification, and/or the rule XYZ to be active under expansion, etc. If we review many *Mathematica* programs, we can see that often there are some very important structure functions that have been written to accomplish what can naturally be expressed using dynamic rules and assignments. Indeed, some simple examples of such functions constitute the subject matter of the next two subsections. Thus, dynamic rules form an extremely important extension to *Mathematica*'s rule base, allowing us much greater expressibility. Just how true this statement is will only become really evident with the work of §5 *Prototypical Structures and Quantum Mechanics* and §7 *Tensor Calculus, Applications, and Quasi-Spin*.

4.6.5 Dynamic Assignment Example: Non-commutative Expansion

Let us now give a simple and semi-useful example of an application of dynamic assignment. Many readers might have felt frustrated with the earlier implementation of non-commutative times, which we introduced in §2.7.4 *Example Calculations from Physics*. Recall that the non-commutative products were *always* expanded, as mandated by the assignments provided in that subsection. For reference, these are repeated below.

```
ℓ___ · (c_?ConstQ a_) · r___ := c ℓ · a · r
ℓ___ · (a_ + b_) · r___ := ℓ · a · r + ℓ · b · r
```

However, a better approach would be to have these "expansion" assignments being "active" only under expansion. This would have better coherence with standard *Mathematica*, since normally products are not expanded by default.

```
In[19]:= a (b + c)

Out[19]= a (b + c)
```

We need `Expand` to expand this product.

```
In[20]:= Expand @ %

Out[20]= a b + a c
```

So too, we would like this same behavior to be true for our `NonCommutativeTimes`. As before, let us first furnish `NonCommutativeTimes` with an infix notation and specify some rules for `ConstQ`.

```
In[21]:= InfixNotation[·, NonCommutativeTimes];
         SetAttributes[NonCommutativeTimes, {Flat, OneIdentity}];

In[22]:= ConstQ[args_Times] := And @@ ConstQ /@ List @@ args
         ConstQ[args_Plus] := And @@ ConstQ /@ List @@ args
         ConstQ[c_^n_] := ConstQ[c] ∧ ConstQ[n]
         ConstQ[_?NumberQ] := True
         ConstQ[_] := False
```

Using dynamic rules, it is now indeed trivial, as claimed, to state that the expansion rules for our `NonCommutativeTimes` are only active within the `Expand` environment.

```
In[27]:= Expand ↗ ℓ___ · (c_?ConstQ a_) · r___ := c ℓ · a · r
         Expand ↗ ℓ___ · (a_ + b_) · r___ := ℓ · a · r + ℓ · b · r
```

Consequently, expansion only occurs through `Expand` instead of being mandatory.

```
In[29]:= a · (b + 2 c)

Out[29]= a · (b + 2 c)
```

```
In[30]:= Expand @ %
Out[30]= a · b + 2 a · c
```

Moreover, expansion works at any level inside an expression.

```
In[31]:= f[g[h[x · (a + b + c) · z]]]
Out[31]= f[g[h[x · (a + b + c) · z]]]

In[32]:= Expand @ %
Out[32]= f[g[h[x · a · z + x · b · z + x · c · z]]]
```

In addition, it respects the holding attributes of the various functions.

```
In[33]:= Expand[Hold[x · (a + b)] + z · (c + d)]
Out[33]= Hold[x · (a + b)] + z · c + z · d
```

In summary, in a very "natural" way we have furnished the operation `NonCommutativeTimes` with what most users would expect to be normal behavior. An alternative approach, the one most commonly used in the past, is to define a separate function, maybe something like `NCExpand` (see for instance FeynCalc[230, 231], NCAlgebra[160], Tracer[174], etc.).

With dynamic rules introduced, we could now progress onto building up a collection of generic structures that are useful in physics. Once this is complete, we will use our inheritance paradigm to endow particular operators and operations with the correct expansion, simplification, factorization, and manipulation routines. However, such work will be deferred until §5 *Prototypical Structures and Quantum Mechanics* and §7 *Tensor Calculus, Applications, and Quasi-Spin.*

Not only can we arrange, via inheritance, how structures work; but we are also able to arrange, via inheritance, how other external operations work on these data structures. This appears to be a key difference between our inheritance model and that offered by, say, abstract data types and/or functors [327, 339].

Technical Note: Given the breadth of the field of object-oriented programming, it is, of course, almost a certainty that there are parallels to our approach that can be drawn with some form or other of inheritance appearing in the literature.

4.6.6 Dynamic Assignment Example: Threading over == (Equal)

To complete this section, let us consider one final simple problem. Fortuitously, Trott[312] and Maeder[222] also address this quandary. Consider the function `Equal`. It is not automatically threaded over certain expressions that one might normally expect it to be. For example, one might expect that adding two equations might result in a new equation with the various sides being added. But that does not happen. For instance

```
In[34]:= (a == b) + (c == d)
```

```
Out[34]= (a == b) + (c == d)
```

We might reasonably have expected the answer to be a+c == b+d. Indeed, if one were asked what should be the result of adding two equations, then the probable answer would be the one just given. So how can we bring about this behavior? One way is to blithely override Equal, as Trott does. (No appropriation of blame should be ascribed to Trott for doing this, since his example was largely intended for illustration, as is our use of the problem.) However, Trott's code enforces all heads to be threaded over Equal whenever possible. So, this "threading over" behavior occurs not just for Plus, but also for every other possible head, like Times. Maeder gives a more refined solution than Trott, but his is still always active. (In certain circumstances this may be the desired behavior.)

Using normal *Mathematica,* we could easily define a function to bring about the desired threading of equals over appropriate heads. However, such threading can be elegantly stated using dynamic rules. Let us proceed to do the latter. The following states that under the environment Over, appropriate heads will be threaded over Equal. That is, we give dynamic assignments for Equal in the environment Over.

```
In[35]:= Over ↗ Equal /: head_[u_ == v_, x_ == y_] :=
           (head[u, x] == head[v, y]) /; distributeEqualOverQ @ head
         Over ↗ Equal /: head_[ℓ___, x_ == y_, r___] :=
           (head[ℓ, x, r] == head[ℓ, y, r]) /; distributeEqualOverQ @ head

In[37]:= Over @ any_ := any
```

This raises the question: What is an appropriate head? The answer is anything with a precedence higher than Equal, but not in the set of "solution type" functions. This is easily implemented as follows.

```
In[38]:= distributeEqualOverQ @ head_Symbol :=
           ¬ MemberQ[headsNotToDistributeOver, head] ∧
             (Precedence @ head ≥ Precedence @ Equal)

In[39]:= headsNotToDistributeOver = {Solve, Reduce, Eliminate, SolveAlways,
             Roots, ToRules, NSolve, DSolve, NDSolve, FindRoot};
```

Now we can easily add equations, and by design, no new behavior is exhibited.

```
In[40]:= (a == b) + (c == d)

Out[40]= (a == b) + (c == d)
```

Yet, if we want our expressions to be threaded over Equal, we accomplish this using the function Over.

```
In[41]:= Over [(a == b) + (c == d)]

Out[41]= a + c == b + d
```

Note, it would not be possible to just use Distribute because this would lead to some nonsensical results. For instance

```
In[42]:= Distribute[(a == b) ^ (c == d), Equal]
```

```
Out[42]= a^c == a^d == b^c == b^d
```

In contrast, when we use `Over`, we get the desired result.

```
In[43]:= Over [ (a == b) ^ (c == d) ]

Out[43]= a^c == b^d
```

Here are some other examples illustrating the use of `Over`.

```
In[44]:= Sin[a == b]

Out[44]= Sin[a == b]

In[45]:= Over @ %

Out[45]= Sin[a] == Sin[b]

In[46]:= x + (a == b)

Out[46]= x + (a == b)

In[47]:= Over @ %

Out[47]= a + x == b + x
```

It should be noted that one could, of course, create the function `Over` using conventional rules without too much difficulty. (All of the original problems given in §4.6.1 *Motivation for Dynamic Rules* would then be encountered.) However, as demonstrated, it is both easy and extremely elegant to use dynamic assignments to solve our "threading over Equals" problem.

As already mentioned, the major applications of dynamic rules to the structures we will use in physics calculations are given later, in §5 *Prototypical Structures and Quantum Mechanics* and §7 *Tensor Calculus, Applications, and Quasi-Spin*. For the remainder of this chapter, we concentrate on the implementation details of the *Assign* package and possible implications, as well as possible further developments.

4.7 Technical Details

This section deals with several of the technical details associated with the *Assign* package. Some simple issues involved in implementing the *Assign* package are discussed in the first subsection. Three new value functions are then introduced and illustrated in §4.7.2 *Attribute Values* and §4.7.3 *DynamicValues and EnvironmentValues*. Following this, a discussion of how dynamic rules and dynamic assignments are implemented is given in §4.7.4 *Underpinnings of Dynamic Transformation Rules* and §4.7.5 *Underpinnings of Dynamic Assignments*. The implementation of these objects being carried out in *Mathematica* itself raises several problems — problems which could be corrected if one had access to the internals of *Mathematica*. These problems and related issues are discussed in §4.7.6 *Caveats about Dynamic Rules I: Renegade Environment Rules* and §4.7.7 *Caveats about Dynamic Rules II: Efficiency*.

4.7.1 General Implementation Issues

The major functions `Values` and `Assign` are generally not extremely difficult to implement. Indeed, there is no real trickiness involved in their definitions. However, the coding details of assigning dynamic rules is non-trivial, consequently the implementation details of this specific case are deferred until a later subsection.

In short, the function `Values` applied to a symbol, say *symb*, basically just collects all the values for *symb* of the various value functions, putting the tag *symb* on each one, and then returns the result. The values returned by the functions `AttributeValues`, `DynamicValues`, and `EnvironmentValues` must be incorporated into the result returned by `Values` with slightly more care. (We describe the new value functions shortly.)

The function `Assign`, as applied to rules and tagged rules, is similarly easy to implement. For instance, any tagged rule, say *tag_* /: *lhs_* → *rhs_*, is simply transformed to the assignment *tag* /: *lhs* = *rhs*. Care must admittedly be taken to ensure that the expressions are held correctly, and other miscellaneous details; but there is nothing intrinsically difficult in this part of the code. For instance, the particular fragment of the `Assign` code which assigns a tagged rule is given in the source code as follows.

```
Assign [ tag_ /: lhs_ → rhs_ ]hp := (tag /: lhs = rhs;);
```

There are several non-trivial notations that are defined by the *Assign* package. To create these, the *Assign* package pivotally uses the *Notation* package, already detailed in §2 *The Notation Package* and §3 *Foundations of Notation*. (Actually, the need to create these notations was one of a handful of original problems that prompted the author to produce the *Notation* package.) The main new notations are for tagged rules, dynamic rules, and dynamic assignments. (The notation for tagged assignments is already handled by *Mathematica*.) The implementation of

the notations is fairly readable. For instance, here is one of the rules governing the parsing of dynamic rules.

```
Notation[
   env_ ↗ tag_ /: lhs_ → rhs_ ⟹ DynamicRule[lhs_, rhs_, tag_, env_]];
```

Yet, there are several places where complex patterns — first discussed in §3.3 *Complex Patterns in Notations* — are needed. For instance, when the tag that a rule should be associated with is omitted, then the tag should be determined by the left hand side of the rule. This is done by the following notation statement.

```
Notation[ env_ ↗ lhs_ → rhs_ ⟹
      DynamicRule[lhs_, rhs_, findTagInBoxes[lhs_], env_]];
```

where `findTagInBoxes` is suitably defined. Similar notation statements are just as easily written for the formatting of tagged rules, dynamic rules, and dynamic assignments. But, as noted earlier, care is taken to obey the option `ShowTags`, which was added to the function `Format`. In fact, if the tag is to be visually omitted in a tagged rule, the formatting routines need to include a tag box in the underlying output structure so when the structure is edited, it remains a tagged rule. It is instructive to look at the box structure of such an example.

```
In[1]:= g /: g[x_] → x^2
```

Out[1]= $g[x\_] \to x^2$

The above output has the following underlying box structure. (It may be insightful to review §3.2.2 *Tag Boxes*.)

```
TagBox[RowBox @{
         RowBox@{"g", "[", "x_", "]"}, " ",
         "→", " ", SuperscriptBox["x", "2"]},
   Assign`Private`taggedStructure,
   StripWrapperBoxes → True]
```
(4.7.a)

The *Assign* package suitably defines the function `taggedStructure` in such a way as to ensure that the output expression, when used as an input expression, is parsed to a tagged rule.

Let us progress onto the special value functions that the *Assign* package sets up.

4.7.2 Attribute Values

To adhere to the paradigm introduced in §4.4 *Assignment and Inheritance*, it is necessary that `Values`, applied to a symbol, returns "everything" to do with that symbol. So how should the information about the attributes be incorporated into "everything"? When one sets some attributes for a symbol, say `HoldAll` and `Listable` for the symbol `foo`, one does it like so.

```
In[2]:= SetAttributes[foo, {HoldAll, Listable}]
```

Clearly, this does not look like an assignment. How then should `Values` incorporate the attribute information now associated with `foo`?

```
In[3]:= ?foo

        Global`foo

        Attributes[foo] = {HoldAll, Listable}
```

Our inheritance paradigm is based on transforming rules produced by `Values` and then assigning the result. Thus, `Values` must return the information about the attributes of the symbol under consideration in the form of rules. The simplest way to accomplish this would be to have `Values` return something like

$$\texttt{Attributes[foo]}_{\texttt{hp}} \rightarrow \texttt{\{HoldAll, Listable\}} \qquad (4.7.b)$$

This would fit our inheritance paradigm perfectly. We could make some replacements and then reassign the resulting rule. For instance, say `bar` inherits the rules for `foo`. Then after the inheritance, `bar` will have the attributes `HoldAll` and `Listable`. Unfortunately, there are problems with this approach, which we soon address. First though, let us demonstrate what would happen if `Values` returned the above form of rules.

```
In[4]:= theFooValues = Attributes[foo]hp → {HoldAll, Listable};
```

Let us create a simple rule for `bar`, and then have `bar` inherit the "proposed values" of `foo`.

```
In[5]:= bar[x_] := x^2

In[6]:= Assign[theFooValues /. foo → bar]
```

Now `bar` has inherited the attributes of `foo`.

```
In[7]:= ?bar

        Global`bar

        Attributes[bar] = {HoldAll, Listable}
        bar[x_] := x^2
```

However, there is a problem with the above approach. Normally `Assign`, acting on $lhs_{\texttt{hp}} \rightarrow rhs$, produces the assignment $lhs = rhs$. Hence, if `Assign` acts on (4.7.b) in its standard way, it will make the assignment `Attributes[bar] = {HoldAll, Listable}`, hence any previous attributes of `bar` will be wiped. Except in special circumstances, this is clearly undesirable. We want `bar` to be able to inherit new information without necessarily loosing pre-existing information. For instance, in the above, `bar` did not loose the rule `bar[x_] := x^2` when it inherited the values of `foo`. Moreover, if existing information about `bar` is wiped when `bar` inherits new definitions, then multiple inheritance from several symbols would not be possible, whereas it is eminently possible.

A possible resolution to the above problem is to modify the behavior of `Assign` when it encounters rules of the form `Attributes[`*symbol*`]`$_{\texttt{hp}} \rightarrow$ {*attributes*}. Instead of the normal behavior, we could have `Assign` translate this rule to the statement `SetAttributes[`*symbol*, {*attributes*}`]`. Then, only the specified attributes of *symbol* would be affected and all other

attributes would remain unchanged. However, with this approach, we would not be able to clear particular attributes of a symbol by rule assignment. We will find in §B.1 *Attributes, Pattern Matching, and Associativity* that sometimes it is desirable, or even necessary, to be able to also remove attributes by rule assignment.

It should therefore be evident that none of the proposed solutions above are entirely adequate. Due to these considerations, we adopt a slightly different approach. We introduce a testing function which, when directly set, results in the side effect of setting or clearing the specified attribute for our symbol.

`AttributeIsSetQ[` *symb*`,` *attr*`]`	returns `True` if *attr* is an attribute of the symbol *symb*, and `False` otherwise
`AttributeIsSetQ[` *symb*`,` *attr*`] = True`	sets the attribute *attr* of *symb*
`AttributeIsSetQ[` *symb*`,` *attr*`] = False`	clears the attribute *attr* of *symb*

The effects of setting the function `AttributeIsSetQ`.

Using the function `AttributeIsSetQ`, `Values` can return a list of tagged transformations encapsulating the state of the given symbol's attributes. Let us demonstrate this.

```
In[8]:= Values @ foo

Out[8]= {foo /: AttributeIsSetQ[foo, HoldAll] → True,
          foo /: AttributeIsSetQ[foo, Listable] → True}
```

In this manner, `Values` incorporates the attributes of a symbol into the rule set it returns. For completion, let us demonstrate inheriting these rules to a symbol, say `goo`, already having attributes.

```
In[9]:= Attributes[goo] = {Orderless};

In[10]:= Assign[Values @ foo /. foo → goo]

In[11]:= ? goo

         Global`goo

         Attributes[goo] = {HoldAll, Listable, Orderless}
```

If a specific prototypical function needs to remove a specific attribute, then this can be easily accomplished by including a rule of the form *symb* `/: AttributeIsSetQ[`*symb*`,` *attr*`] → False` in the prototypical rules.

To specifically access the attribute values, one uses the *Assign* package function `AttributeValues`, which returns the rules associated with the attributes.

`AttributeValues [` *symb*`]`	returns a list of transformation rules corresponding to all attributes defined for the symbol *symb*.

The attribute values function.

To illustrate `AttributeValues`, consider the following.

```
In[12]:= AttributeValues @ bar

Out[12]= {bar /: AttributeIsSetQ[bar, HoldAll] → True,
           bar /: AttributeIsSetQ[bar, Listable] → True}
```

As with the other value functions in *Mathematica,* it is possible to set the attribute values directly. For instance,

```
In[13]:= AttributeValues @ bar = {bar /: AttributeIsSetQ[bar, HoldAll] → True,
             bar /: AttributeIsSetQ[bar, OneIdentity] → True}

In[14]:= ?? bar

         Global`bar

         Attributes[bar] = {HoldAll, OneIdentity}
         bar[x__] := x^2
```

Technical Note: The attribute values are returned first in the list of all values. The reason for this is that any rule which makes use of a specific attribute must be assigned after the attribute is set or else the rule will not function correctly. This is a feature that is fundamental to *Mathematica* and has nothing to do with the *Assign* package. (Extremely technical comment: some users might actually make use of this feature in *Mathematica* and define certain rules before they set an attribute in order that these rules will *not* make use of the attribute. Only later will they then set the attribute. Thus, if users want to make use of this rather unorthodox and possibly dangerous style of programming, then they have to edit the order in which the rules are returned from the values function. More is said on this in §B.1 *Attributes, Pattern Matching, and Associativity,* but largely this topic lies outside the scope of this chapter.)

In the next subsection we consider, in succession, the other value functions that are defined by the *Assign* package.

4.7.3 DynamicValues and EnvironmentValues

Since we now have dynamic assignments, it follows that we must also correspondingly have dynamic values, hence should have some new corresponding value functions. This subsection describes such functions.

`DynamicValues[` *symb*`]`	returns the list of dynamic transformation rules corresponding to all dynamic assignments associated with the symbol *symb*.
`EnvironmentValues[` *symb*`]`	returns the list of dynamic transformation rules corresponding to all dynamic assignments having the environment *symb*.

The dynamic and environment values.

DynamicValues and EnvironmentValues have complementary aspects. DynamicValues deals with dynamic rules from the perspective of tags, whereas EnvironmentValues deals with dynamic rules from the perspective of environments. We will even speak of *environment rules*, meaning dynamic rules with a particular environment.

To illustrate these functions, let us first re-enter the rules that we previously used for NonCommutativeTimes.

```
In[15]:= InfixNotation[·, NonCommutativeTimes];
         SetAttributes[NonCommutativeTimes, {Flat, OneIdentity}];
```

Now, let us also re-enter the simple dynamic expansion rules for NonCommutativeTimes objects.

```
In[16]:= Expand ↗ ℓ___ · (c_?ConstQ a_) · r___ := c ℓ · a · r /; ({ℓ, r} =!= {})
         Expand ↗ ℓ___ · (a_ + b_) · r___ := ℓ · a · r + ℓ · b · r
```

Further, let us add another rule for our NonCommutativeTimes operation.

```
In[18]:= Simplify ↗ ℓ___ · 0 · r___ = 0
Out[18]= 0
```

This rule should unquestionably be active all of the time, but for illustration's sake, let us assume we only want it true for simplification.

In addition to the above rules for non-commutative times, for illustration purposes, we need dynamic rules for another sort of operation. We choose algebraic summation. We re-introduce the notations for algebraic sums and abstract functions, as treated in §2.2.2 *Notation: Examples*.

```
In[19]:= Notation[ Σ_indices__ sum_ ⟺ AlgebraicSum[sum_, indices__]]
         Notation[body_&λ__ ⟺ Function[{λ__}, body_]]
```

Let us also quickly give one of the rules for the expansion of algebraic sums.

```
In[21]:= Expand ↗ (Σ_indices__ arg_Plus) := (Σ_indices λ &λ) /@ arg
```

AlgebraicSum now incorporates a small part of what one would expect to be the normal behavior of an algebraic sum, in that it behaves correctly under expansion.

```
In[22]:= Σ_{α,β} (f[α, β] + g[α, β])
Out[22]= Σ_{α,β} (f[α, β] + g[α, β])

In[23]:= Expand @ %
Out[23]= Σ_{α,β} f[α, β] + Σ_{α,β} g[α, β]
```

Later, in §5.4.8 *Algebraic Sums*, we give the other rules that govern the behavior of our algebraic sums. For now though, we have entered enough rules to be able to illustrate both `DynamicValues` and `EnvironmentValues`. Let us display the dynamic rules for our operation ·, that is, all the dynamic rules for `NonCommutativeTimes`.

```
In[24]:= DynamicValues[NonCommutativeTimes]
```

```
Out[24]= {Expand ↗ ℓ___ · (a_ c_?ConstQ) · r___ :↦ c ℓ · a · r /; {ℓ, r} =!= {},
           Expand ↗ ℓ___ · (a_ + b_) · r___ :↦ ℓ · a · r + (ℓ · b) · r,
           Simplify ↗ ℓ___ · 0 · r___ :↦ 0}
```

This returns all the dynamic rules associated with `NonCommutativeTimes`. Notice that the dynamic transformation rules can have different environments, since we are requesting all dynamic rules associated with the symbol `NonCommutativeTimes`, rather than all dynamic rules with a given environment. This is the purpose of `DynamicValues`. If, however, we wanted all dynamic rules having a given environment, we could use `EnvironmentValues`.

```
In[25]:= EnvironmentValues[Expand]
```

```
Out[25]= {Expand ↗ ∑_{Indices__} arg_Plus :↦ (∑_{Indices} λ &_λ) /@ arg,
           Expand ↗ ℓ___ · (a_ c_?ConstQ) · r___ :↦ c ℓ · a · r /; {ℓ, r} =!= {},
           Expand ↗ ℓ___ · (a_ + b_) · r___ :↦ ℓ · a · r + (ℓ · b) · r,
           Expand ↗ e /: e^(a_+b_)^n_ :↦ e^Expand[(a+b)^n]}
```

Again, it is instructive to note that the environment rules need not all be associated with the same tag. One environment can have rules for many different operations.

The dynamic and environment values for a symbol can be directly set, just like the attribute values and other types of *Mathematica* values. For example, if we want to set our factorization rules to be those of our expansion rules, we could simply do this as follows.

```
In[26]:= EnvironmentValues[Factor] :=
           EnvironmentValues[Expand] /. Expand → Factor
```

And if we wanted to clear our factorization rules, that is, our rules having the environment `Factor`, we could do this by the following.

```
In[27]:= EnvironmentValues[Factor] =.
```

Instead of directly setting the values functions, it is generally preferable to inherit rules. That way, any previous assignments are unaffected. We perform this inheritance, as always, using `Assign`.

```
In[28]:= Assign[EnvironmentValues[Expand] /. Expand → Simplify]
```

Now, not only does `Simplify` keep all its original dynamic rules, but in addition it has also inherited the dynamic rules of `Expand`.

```
In[29]:= Simplify[∑_{α, β} (f[α, β] + g[α, β])]
```

```
Out[29]= Sum[f[α, β], {α, β}] + Sum[g[α, β], {α, β}]

In[30]:= EnvironmentValues[Simplify]

Out[30]= {Simplify ↗ e /: e^(a_+b_)^n_ :↦ e^Simplify[(a+b)^n], Simplify ↗ ℓ___ · 0 · r___ :↦ 0,
    Simplify ↗ ℓ___ · (a_ c_?ConstQ) · r___ :↦ c ℓ · a · r /; {ℓ, r} =!= {},
    Simplify ↗ ℓ___ · (a_ + b_) · r___ :↦ ℓ · a · r + (ℓ · b) · r,
    Simplify ↗ Sum[arg_Plus, {indices___}] :↦ (Sum[λ &λ, {indices}]) /@ arg}
```

4.7.4 Underpinnings of Dynamic Transformation Rules

Obviously, since we have altered *Mathematica* in a substantial way, the inevitable question arises: How was this done? The short answer is dynamically. Again, as in earlier sections, the implementation details will be covered in a somewhat cursory fashion. In this subsection, we will examine some of the implementation details of dynamic transformation rules, while the sequel on dynamic assignments will be given in the next subsection.

First, let us examine how a dynamic rule is translated into its corresponding normal replacement rule. (We do this in the `Assign`Private`` context to ease readability.)

```
In[31]:= Begin @ "Assign`Private`";
         ClearAll @ goo;
         toReplacementRules [goo ↗ e /: (e^a_)^b_ :↦ e^a b]

Out[33]= expr$ : goo[___]_hp :↦ With[{evaluate$ = assignHold[expr$] /. (e^a_)^b_ :↦ e^a b},
             evaluate$ /; evaluate$ =!= assignHold[expr$]]
```

It is doubtful whether an explanation of the above code fragment would provide more insight than does the code itself. But in general terms, the dynamic rewrite rule is transformed into a replacement rule that, when used, looks for expressions inside the environment under consideration. Let us illustrate this particular rule acting on an expression.

```
In[34]:= Hold[goo[x + (e^a)^b + 2 + 2] + (e^i)^j] /. %

Out[34]= Hold[assignHold[goo[x + e^a b + 2 + 2]] + (e^i)^j]
```

The function `assignHold` is required in order to correctly maintain the same kind of evaluation mechanism under replacement as *Mathematica* has. For comparison, note the following standard behavior of *Mathematica*.

```
In[35]:= Hold[goo[x]] /. goo[x] :↦ 2 + 2

Out[35]= Hold[2 + 2]
```

For the sake of comparison, note that by design, almost the exact same thing happens with dynamic rules.

```
In[36]:= Hold[goo[x]] /. (goo ↗ x :↦ 2 + 2)

Out[36]= Hold[goo[2 + 2]]
```

This coherence with standard *Mathematica* replacement is the reason why we require that the `assignHold` wrapper is present in the transformed rules. In fact, we can see that if we were to remove this wrapper, then inadvertent evaluation would sometimes take place. To confirm this, let us remove the `assignHold` wrapper from the transformed rule of `goo ↗ x :↦ 2+2`, and retry our above example.

```
In[37]:= Hold[goo[x]] /. expr$ : goo[___]_hp :↦
           With[{evaluate$ = expr$ /. x :↦ 2 + 2}, evaluate$ /; evaluate$ =!= expr$]

Out[37]= Hold[goo[4]]
```

The introduction of the `assignHold` wrapper unfortunately requires that we must correspondingly override the functions `Replace`, `ReplaceAll`, and `ReplaceRepeated` so that they remove the wrapper once replacement is complete. Both `Replace` and `ReplaceAll` are modified in such a way as to incorporate a `releaseAssignHold` that removes the `assignHold` added in any replacement. Here is the definition of this override for `Replace`.

```
Replace[expr_, rules_?containsDynamicRulesQ] :=
  releaseAssignHold @
    Replace[expr, Evaluate @ toReplacementRules @ rules];
```

However, slightly more care must be taken with the case of `ReplaceRepeated`.

```
ReplaceRepeated[expr_, rules_?containsDynamicRulesQ] :=
  With[{theRules = toReplacementRules @ rules},
    FixedPoint[
      releaseAssignHold @ ReplaceAll[#, theRules] &, expr]];
```

As dictated by the above code, the constructed replacement rule is applied and then the `assignHold` is released. This is repeated until no further change is possible. Unfortunately, the evaluation semantics might differ from that of normal *Mathematica*. We are implicitly assuming that `ReplaceRepeated` is equivalent to repeated replacement, or more formally, that the following statement is always true for reasonable instances of *expr* and *rules*.

```
ReplaceRepeated[expr, rules] ≡ FixedPoint[(# /. rules) &, expr]
```

Since there is no publicly available exact specification for the semantics (behavior) of `ReplaceRepeated`, and indeed since to the best of the author's knowledge there is no specification documented even within WRI, then it is likely that the above prescription is not, in fact, always true. Due to the substitution of a fixed point structure for a `ReplaceRepeated` structure, we can unfortunately, at times, inflict more than an order of magnitude decrease in speed in the overall replacement process.

Technical Note: There might exist a rewrite rule that one could construct from the dynamic rule which actually achieves the desired holding attributes without introducing a wrapper (such as `assignHold`) that must be present and then removed at a later stage. The author has tried to find such a rule by manipulating various combinations and interactions of `With`, `Block`, `Unevaluated`, etc. Unfortunately, no such combination was found.

4.7.5 Underpinnings of Dynamic Assignments

The underpinnings of dynamic assignments are more involved than those of dynamic transformation rules. For clarity, we continue to operate within the context `Assign`Private`.

The main mechanism whereby dynamic assignments operate is by using the dynamic properties of a `Block` statement. The functions that have environment values must be overridden in order to effect the behavior we have previously illustrated. The overriding of a given function is achieved with a simple conditional pattern dependent on a variable that is dynamically changed. For instance, from the previous examples in this section, `Expand` has some environment rules. We can view the rule that forces the overriding of `Expand` by examining the downvalues of `Expand`.

```
In[38]:= DownValues @ Expand

Out[38]= {(Expand[expr$___] /; dynamicExpandRulesInactive)  :>
              evaluateExpandWithDynamicRules[expr$]}
```

Technical Note: The override definition for the environment assignments will not show up in the information on a symbol, for instance `??Expand`. This is intentional in that the *Assign* package modifies the function information to include the dynamic assignments, but explicitly hides the override definition since it is an implementation part of the language modification.

The symbol `dynamicExpandRulesInactive` is initially set to true and henceforth is never changed by the *Assign* package.

```
In[39]:= dynamicExpandRulesInactive

Out[39]= True
```

However, *within* the function `evaluateExpandWithDynamicRules`, `dynamicExpandRulesInactive` will be dynamically set to false. Consequently, the override rule will only be called once. This is a somewhat typical programming structure allowing a single override and can be found in many of the books and articles detailing programming with *Mathematica* — for instance, Trott[312], Maeder[222], and Wagner[325].

If we examine the definition of `evaluateExpandWithDynamicRules`, we would find almost the following (modulo cosmetic reformatting).

```
evaluateExpandWithDynamicRules[expr___] :=
   Block[
      {ans, dynamicExpandRulesInactive = False, wasProtected},
      wasProtected = Unprotect @
            Evaluate @ tagsOfEnvironmentRules @ Expand;
      Assign @ environmentRules @ Expand;
      ans = CheckAbort[ Expand @ Evaluate @ expr,                    (4.7.c)
```

```
        (UnAssign @ environmentRules @ Expand;
           Protect @ Evaluate @ wasProtected; Abort[])];
    UnAssign @ environmentRules @ Expand;
    Protect @ Evaluate @ wasProtected;
    ans];
```

First, we can see that `dynamicExpandRulesInactive` is dynamically set to false. Second, we record which tags are protected and then unprotect these tags. We must do this if we are making a dynamic assignment that is associated with a tag that is protected. For instance, the dynamic rule `Expand` $\nearrow$ e /: $(e^{a}\_)^{b\_} :\mapsto e^{a\,b}$, has the tag e, yet e is protected. Third, we assign the environment values of `Expand`, hence the environment rules are now "active". Fourth, we perform the actual expansion, checking to make sure that if the calculation is aborted, we do it cleanly. This calculates the value while the environment rules are active, and we record the returned answer. Fifth, we deactivate the environment rules. Sixth, and lastly, we re-protect the functions that we unprotected and return the recorded answer.

Of course the above example, where we used `Expand`, carries over more generally to every other environment. For example, if we had used the environment `Foo`, then our overriding condition would have been denoted `dynamicFooRulesInactive`, and the overriding function would have been called `evaluateFooWithDynamicRules`, etc.

We can summarize the overall process. When a function with environment rules is encountered, we temporarily override the function, then assign the environment rules, then perform the calculation and record the answer, then unassign the rules, and finally return the answer. It is actually rather simple, but it all hinges on *Mathematica*'s ability to create and remove rules dynamically.

Of course, rather complex construction code is involved in order that functions like `evaluate-ExpandWithDynamicRules` can be created on the spot (dynamically) whenever a new dynamic assignment is entered. Again, just as in the case of the *Notation* package, for further details the interested reader should consult the source code for the *Assign* package.

Before progressing, let us restore the context to the global context.

```
In[40]:= End[]

Out[40]= Assign`Private`
```

Elegant though it is, there are still some problems with the above approach to dynamic assignments that we have managed to graft onto *Mathematica*. Let us next examine some of these difficulties.

4.7.6 Caveats about Dynamic Rules I: Renegade Environment Rules

There are a few limitations in our working model of dynamic assignments that are not practical to overcome. The first is that environment rules can unfortunately remain active outside their intuitive scope, owing to the `Block` used to implement the assignment. Here is an illustrative example.

```
In[41]:= ClearAll[f, goo];
         goo↗x := 2
         goo @ any_ := f @ any
```

Now goo has dynamic values, as expected. For instance:

```
In[44]:= goo[x]

Out[44]= f[2]
```

This is the expected behavior. However, what happens when we create a new rule for `f` which involves x, and we use it in conjunction with goo?

```
In[45]:= f @ any_ = x;
```

Let us examine the result of the following input.

```
In[46]:= goo[x]

Out[46]= 2
```

This is unexpected. Normal intuition dictates that the evaluation sequence would progress as follows.

$$\texttt{goo[x]} \xrightarrow[\text{of x under goo}]{\text{dynamic evaluation}} \texttt{goo[2]} \xrightarrow[\text{of goo}]{\text{standard evaluation}} \texttt{f[2]} \xrightarrow[\text{of f}]{\text{standard evaluation}} \texttt{x} \tag{4.7.d}$$

How, then, does x evaluate to 2? It transpires that the dynamic rules for x under the environment of goo persist, even after the head goo is gone. This is not too hard to understand, since we implemented the dynamic rules using a `Block`. The dynamic rules will remain active throughout the specific calculation that spawned them until the scope of the `Block` ends, regardless of whether the environment head that spawned the dynamic evaluation still exists. Thus, the evaluation actually proceeds via the following:

$$\begin{aligned}\texttt{goo[x]} &\xrightarrow[\text{of x under goo}]{\text{dynamic evaluation}} \texttt{goo[2]} \xrightarrow[\text{of goo}]{\text{standard evaluation}} \\ \texttt{f[2]} &\xrightarrow[\text{of f}]{\text{standard evaluation}} \texttt{x} \xrightarrow[\text{of x under goo}]{\text{dynamic evaluation}} 2\end{aligned} \tag{4.7.e}$$

Intuitively, this is irritating, since one might feel that the goo environment is no longer present. There is no environment head left to form the "environment", yet the rules persist throughout

the evaluation process like some blind rudderless vessel, charging on regardless. Certainly if one had access to the internals of *Mathematica,* one could, and would, properly rectify this flaw. Ideally, dynamic rules would only be active within their given environment. What to do? To resolve this problem, we introduce the functions `DynamicBar` and `ReleaseDynamicBar`.

`DynamicBar [` *expr*`]`	acts as a wrapper, holding all evaluation of *expr* until released by a `ReleaseDynamicBar` . It can be used in the right hand sides of definitions involving any function that has environment values in order to limit the scope of that function' s environment values.
`ReleaseDynamicBar [` *expr*`]`	releases all occurrences of `DynamicBar` that appear within *expr*.

Functions to limit the scope of environment rules.

Technical Note: We might have called this wrapper `DynamicBlock`, to be suggestive of blocking any dynamic values escaping the confines of their scope. However, `DynamicBlock` could be associated with the function `Block`, so it was rejected as a choice of name.

The wrapper `DynamicBar` is just an inert wrapper that has the attribute `HoldAllComplete`. Its complement is `ReleaseDynamicBar`, which releases the wrapper. These functions are much like their compatriots, `Hold` and `ReleaseHold`.

Instead of defining `goo @` *any_* `:= f @` *any*, if we add a `DynamicBar` to the head of the right hand side, this will restrict the environment rules of `goo` from interacting with the evaluation of the right hand side. To demonstrate

```
In[47]:= goo @ any_ := DynamicBar @ f @ any
```

Now re-evaluating our previous computation returns the expected answer.

```
In[48]:= goo[x]

Out[48]= x
```

Why does this work? Because the code for our dynamic rules actually contains a `ReleaseDynamicBar`, to overcome just such a problem. Unfortunately, to solve the problem of dynamic rules escaping their scope once the environment has been transformed away, we have had to manually add a wrapper to the right hand side of the definition which transformed away the environment head.

Technical Note: The question arises of whether the wrapper `DynamicBar` could be added automatically? The answer is Yes, but it might lead to other unwanted problems. We could define "everything" in such a way that as soon as a symbol *symb* has environment rules attached to it, we automatically add a `DynamicBar` to the right hand side of every definition involving *symb*. It would not be particularly hard to do this, but it may have unforeseen ramifications. Such a change would be disquieting and has been forgone. There are questions about upvalues involving the environment symbol, and other boundary cases. If we do add `DynamicBar` wrappers automatically, then do we include the wrappers in the values functions? Do we include them in the information? These issues point to leaving the use of `DynamicBar` up to the user.

In summary, we have surmounted the problem of the renegade environment rules persisting outside their intuitive scope, but at the expense of added programming complexity. As already

mentioned, if one had access to the internals of *Mathematica*, this could be overcome, and ideally it should. However, in practice the user must use the `DynamicBar` wrappers when necessary.

Let us now go on to look at the second problem / caveat with dynamic assignments.

4.7.7 Caveats about Dynamic Rules II: Efficiency

Due to the implementation model of dynamic rules, there is one further caveat that users should be aware of when using them. Assigning and unassigning lists of rules takes time; so if the dynamic rule head is issued in a function many times, for instance something like `Table[Expand[a_i], {i, 1, 10000}]`, then, necessarily, the execution time will be comparatively long, since the expansion rules will be added and removed 10000 times. Such delays can amount to an inordinately long increase in execution. However, something like `Expand[Table[a_i, {i, 1, 10000}]]` would execute without any real decrease in speed.

The above difficulty is not an intrinsic limitation of dynamic rules; rather it occurs due to the fact that the implementation cannot access the internals of the *Mathematica* engine. If one could access the internals, it would be relatively easy to implement dynamic rules so this limitation never occurred. Basically, when the environment function is overridden, we would copy the rules and create a new environment function. Instead of doing this each and every time the environment is used, we could instead create an intelligent caching system along the following lines: once we have created these rules in the given environment for the first time, they would be cached and thus we would avoid recreating them the next time the function is called. Unfortunately, this all requires core access to *Mathematica*'s internals.

To make the above comments more understandable, let us consider a dynamic rule and a translation of it into a piece of pseudo-code. The dynamic rule is

```
foo ↗ f[a_] = b
```

This can be translated into the following pseudo-code.

```
privateF[a] = b

foo[args__] := Block[{f = privateF}, foo[args]] /; executeOnlyOnce
```

Thus, any occurrence of `foo` would automatically use the *privateF* inside its execution. Of course, the above code will not execute in *Mathematica*, and there are deep and fundamental problems with it. However, it should give the idea of what we would do if we had access to the internals of *Mathematica*. We would make private functions that contained all the necessary definitions that we would like to be dynamic. Then we would use these private functions in place of our normal functions in the evaluation of the expression under consideration. There is nothing intrinsically stopping this implementation in *Mathematica*.

Technical Note: The above ideas could be stretched to make some sort of working version in top level *Mathematica*. However, there are several intrinsic problems. Among them are the existence of functions not associated with `f` that hold their values. For instance, assuming we are making use of the rule `h[f[1]]` → λ, then this rule would never fire if we made the replacement `f` →*private*`F`. This too could be rectified by gaining access to the underlying internals of *Mathematica*. But this just reaffirms that we need access to the internals of *Mathematica* in any case. Also, there are other problems/complications with the above approach.

The *Assign* package and its dynamic rules are still very useful, even with the unfortunate limitation in the assign and unassign overhead. Often in language design, the fundamental limits should be probed, but it is beyond the scope of this thesis to create a fully optimized package version. Maybe such work will be undertaken by Wolfram Research. It should not be inordinately difficult.

To reiterate, we will apply these language modifications in §5 *Prototypical Structures and Quantum Mechanics* and §7 *Tensor Calculus, Applications, and Quasi-Spin*. Dynamic rules will be used throughout the rest of this thesis when coding functions.

4.8 Conclusions and Implications

In this section we briefly discuss some of the potential developments before concluding this chapter.

4.8.1 Potential Further Developments

The material of this subsection is more speculative than anything presented so far in this chapter. In this subsection I outline a paradigm for a new language based on collections of rewrite rules. I have not seriously pursued either the compilability of such a language or indeed its efficiencies. However, I would like to outline it, for I believe it would solve some of the problems faced by language designers in symbolic computation.

In the Maple language[57, 244, 329], by default, evaluations do not necessarily take place. To produce evaluation, one must liberally sprinkle `eval` wrappers throughout procedure code. In *Mathematica*, everything by default is evaluated, so we sometimes must be very careful to prevent evaluation. In such cases, `Hold` and `Unevaluated` wrappers must be sprinkled throughout the code. To do this correctly can sometimes be difficult — see Villegas[323]. Somehow one expects that a user should be able to choose any position at or between these opposite extremes at any stage.

With the developments of the previous sections and the advent of dynamic rewrite rules, it is clearly possible to develop a model language that is not viewed as static. However, it would be possible to internally transform the language to an equivalent semi-static language that is largely cached. Thus, it appears that it would be possible to construct a language where the program changes with context. This would be extremely helpful for evaluation. For instance, if

we would like to suspend all evaluation of data structures and only have replacement rules, part rules, and `MyOperatorRules` working, then we could do the following.

```
EvaluationEnvironment[{evaluationRules →
        {ReplacementRules, PartRules, MyOperatorRules}},
  exprToEvaluate]
```

In such a situation, no other evaluations would slow the calculation down or force one to clutter data structures with `Hold` or `Unevaluated` wrappers. Also, it is likely that having a smaller rule set would make dynamic compilation easier, although this might not always be the case. Certainly, if some of the rule structures that necessitated state and voided referential transparency were removed, then compilation would be *much* easier.

Related to the previous discussion, instead of stripping the environment in dynamic rules when we want them to always be active, it might be nice to have a generic environment that overarches everything. Thus, we could express things akin to the following.

```
Always ↗ lhs := rhs
```

The other potential evaluation scheme that I have contemplated would be to give all rules a precedence value. A rule would be executed according to its precedence. For instance, consider

```
300 : f[x_] := 2 x^2
200 : g[x_] := 2 x - 1
```

A typical calculation might then proceed as

```
f[g[x]] → 2 g[x]^2 → 2 (2 x - 1)^2
```

In this way, if one wanted an upvalue, or something like it, one would only need to give the rule a precedence above the rule surrounding it. Also, if one needed to write code to modify other code, all one would have to do is set the precedence of the code transformation rules higher than any of the precedences of the code to be modified, hence one would not have to worry about any inconveniences with holds or evaluates, etc. Indeed if rules had precedences, we could possibly allow conflicting rules by using the precedences of the various rules as their tiebreaking criteria. That is, say the conflicting rules lead to a loop as follows.

$$expr \xrightarrow{\text{Rule } A_1} expr_1 \xrightarrow{\text{Rule } A_2} expr_2 \xrightarrow{\text{Rule } A_3} \dots \xrightarrow{\text{Rule } A_{n-1}} expr^{n-1} \xrightarrow{\text{Rule } A_n} expr$$

Then we could just break the evaluation at the rule A_i which has the lowest precedence of all of the n rules.

The other strong advantage of having rule precedences is that it then would not matter in which order some of the critical rules are entered. For instance, in §5.4.6 *Aside: Commutation of Operators Raised to Powers* we improve upon certain relations, and we would like these new optimized relations to work before the original relations. At present, the process to ensure that the optimized relations are processed prior to the general relations is not as elegant as it could be.

If we were really open to speculation, we could consider modifying the values of the relative precedences of the rules dynamically. Indeed, this would be desirable when we have code that modifies other code. In this way, we would be able to examine the precedences of a given set of rules and then create rules at other precedences, in order to facilitate our goals.

All of these ideas could and should be pursued in the context of languages for symbolic computation. One can say that in the future, the languages for symbolic computation should have the following properties: (*i*) a functional style, (*ii*) graph-rewriting[77, 264] rather than term rewriting[90], (*iii*) a significant degree of compilability, and (*iv*) some degree of typing, probably more than *Mathematica* has but less than, say, Axiom[177] currently has.

Actually, let us briefly discuss an aside about typing in symbolic computation. There has been, of course, much work on this subject — see Temperini and colleagues[91, 204, 64], Cardelli[1, 54], Santas[281, 282], Weber[332], Aldor[31] (the underlying language used by AXIOM[177]), among others. At the very least, any new language for symbolic computing should be a lot less "regimented/inflexible/rigid" in its typing mechanisms than is AXIOM. However, it should be stressed that the typing must somehow be combined with expression simplification. Thus, simple typing constructs like, say, a positive integer, are inherently rather useless, or at the very least extremely non-general. One must have classes of types that are totally flexible. For instance, under limited systems, such as Macsyma[200, 217], the question is sometimes asked: Is the following variable positive, negative, or zero? This is patently unhelpful if one knows, say, that a variable is real and ranges from -1 to 1.

The above raises the question of type coercion in some languages. For instance, AXIOM has the type PositiveInteger. Again, for any generalizable system, such an approach is somewhat doomed. Of course AXIOM has an advanced typing system, but it is not clear to the author when typing information like this is particularly useful. Certainly, in terms of solving equations and inequalities, it is highly likely to be almost useless. In fact, worse than useless, since it obviously encumbers the whole typing system because coercions, etc., must then be made. Thus, a more advanced system of typing is needed. For instance, the mathematical precursors of such a system for real numbers can be found in the theory and algorithms for quantifier elimination[72, 73, 74, 337]. Strzebonski, who at the time of writing works at WRI, has implemented and continues to refine such algorithms for *Mathematica*. This, however, is blurring the line between typing and structures/expressions. Is a type a system of inequations? The question of how this relates to other typed structures — say a group, or a ring, or a differential manifold, or indeed a Hilbert space — presents an open-ended dilemma.

Assuredly, the final word has not been said on programming languages for symbolic computation.

4.8.2 Conclusions

Just as the inheritance paradigm worked with tagged rules and assignments in §4.4.2 *Inheritance*, the same paradigm works with dynamic rules and assignments. Instead of giving somewhat contrived examples of this inheritance, we defer examples of this until §5 *Prototypical Structures and Quantum Mechanics* and §7 *Tensor Calculus, Applications, and Quasi-Spin*.

In conclusion, the *Assign* package gives us true dynamic rule inheritance. We can transform structures dynamically. The mechanisms are very simple and are intuitive for anyone familiar with *Mathematica*. Thus, the *Assign* package allows the adoption of a style of controlled self modifying code (more correctly, self creating).

The inheritance paradigm is similar to object-oriented programming[125, 38, 236]. It is a loose generalization sharing traits from prototype based object-oriented programming[24] and abstract data types[125, 149, 339] in imperative languages, as well as rule based programming in functional languages. It also shares similarities with the goals of generic programming[14, 175]. It might be a stretch to classify it as embodying abstract data types since *Mathematica* is a manifestly untyped language. However, the real inheritance is not of objects but "code", even though these are viewed as one and the same in most object models. Still, the end result of inheriting code gives new code that manipulates structures and expressions. These structures and expressions are not really objects in the classical sense. The working paradigm could possibly be viewed, at a stretch, as "message passing", but it would be mischievous to do so.

Technical Note: The genericity described above, say that of the extension PolyP[175] to Haskell, is to a large extent incorporated into *Mathematica* since the latter is manifestly untyped.

We have achieved the indicated simplicity by unifying the way *Mathematica* handles "values". This unification was accomplished by introducing the notion of tagged transformation rules, which are normal rules together with a tag with which the rule should be associated. `Assign` (`UnAssign`) will just assign (remove) these tagged transformation rules to (from) their associated tag symbols. This refines the way *Mathematica* handles rules into a more integrated and systematic paradigm.

Once the concept of tagged rules, and adding and removing rule sets dynamically, is introduced, the evolution to dynamic rules and dynamic assignments is natural. Dynamic rules and assignments elegantly and beautifully allow the simplistic expression of rules being active in a given environment.

Since the inheritance mechanisms are so trivial to someone who knows *Mathematica*, one does not have to explain the concept of multiple inheritance, overriding, or the other common concepts that occur in object-oriented programming. These are just natural consequences or specific cases of our framework. The user can set up the style of inheritance that best suites his / her application. If the user understands the use of `'/:'` in rules and understands the simple function `Assign`, then there is effectively no learning curve. These tools are completely sufficient for inheritance.

As we have seen in the above examples, inheritance can be as simple as the one line operation `Assign @` *Replacements* `@ Values @` *Symbols*. By design, this working paradigm functions perfectly with dynamic rules and dynamic assignments. Thus, the inheritance of operations will collectively work with a variety of external functions, in a single unified way. In summary, inheritance is encapsulated in the following more general version of Figure 4.4.C.

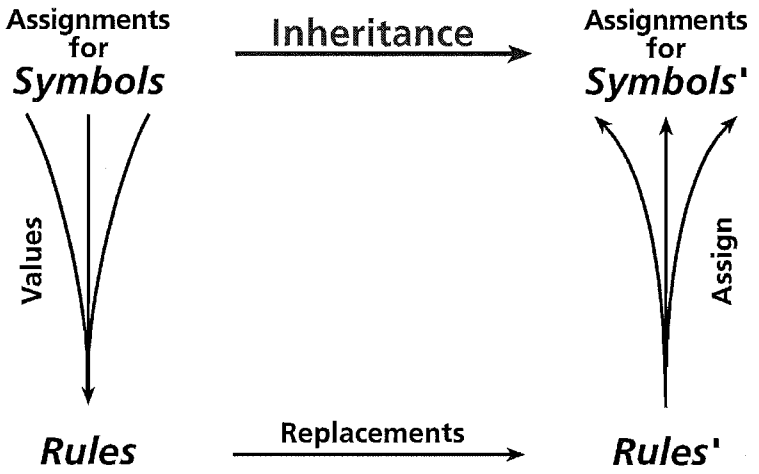


Figure 4.8.A: The process of inheritance.

Of course, due to the fact that the *Assign* package is implemented in top level *Mathematica*, it is a *fait accompli* that using the package will not always lead to the most computationally efficient method of coding a problem. However, it greatly simplifies the coding process. As many users of *Mathematica* and symbolic computation systems in general will be aware, much of symbolic computation is either (*i*) finding specific algorithms to solve specific problems or (*ii*) being able to succinctly and elegantly present and adapt algorithms to the user's specific needs. The paradigm presented in this chapter fulfills the latter goal. Moreover, given access to the internals of *Mathematica* there is no intrinsic reason why a programmer should not be able to modify the internal code in such a way as to add dynamic rules at no extra computational efficiency cost.

The elegant working paradigm developed in this chapter will consequently be adopted throughout the rest of this thesis.

Chapter 5
Prototypical Structures and Quantum Mechanics

5.1 Introduction

5.1.1 Overview

In §2 *The Notation Package* and §3 *Foundations of Notation* we presented the *Notation* package and the notational systems which allow us to accurately represent the objects with which we need to work. In the last chapter, §4 *Language Modifications*, we developed the language extensions that will be used as a basis and framework for our computations. We have now progressed to a stage where it is possible to describe the abstract classes we will use for computations. These classes meld the topics covered so far. This chapter will develop generic classes for use in our physics applications and explain the design choices made to facilitate their use. In the previous chapters, we encapsulated the functionality presented therein into packages. Throughout the course of this chapter, we develop the package *Prototypes* as a by product of our explorations.

In quantum mechanics and other areas of physics and indeed mathematics, we need to be able to express computational properties of our systems. For instance, we might need to specify that something is a linear operator. This could include the information that under expansion linear operators do such and such, or that under simplification all linear operators factorize like so. Another property might be that the operators A and B "normal order" like so under the operation C. We would like a "generic" way of expressing some of these properties in our system.

Of course, in standard *Mathematica* we can usually create a translation from the "semantics" of our example to a set of rewrite rules. The issue we would like to tackle is that of the inheritance and reuse of such structures. By formulating computations in our inheritance paradigm, we obtain a useful level of abstraction and make our system more intelligible. The layer of abstraction also allows us to alter the underlying rewrite rules that our system depends on in order to speed up our computations, yet at the same time modifying just the prototypes that contain our generic code. We have the foundations to pursue this due to the previous work in the §4 *Language Modifications*.

There are two possible approaches we could adopt: (*i*) we could just state, in a postulatory fashion, a large collection of optimized prototypical rules and functions in their most general setting, and then proceed to apply them to specific examples, or (*ii*) we could present generic structures as we need them, finally culminating in a working set of prototypes. There are various advantages and disadvantages to both approaches. We shall adopt the latter approach in all but the first section of this chapter.

Our motivation for creating the prototypes for handling structures in a non-commutative setting and quantum mechanics in particular is that such handling is not currently performed in *Mathematica* [342, 343]. Of course *Mathematica* has a data type for non-commutative multiplication, but it is largely inert. The same is largely true of Maple[244] as well, since even though Maple has a special package for this purpose, its use is problematic. Indeed, Horbatsch [163] goes so far as to state the following.

> ... *Maple has a built in package to deal with non-commuting algebras. We have not figured out how to use it for our purposes in a straightforward way...*

There are certain systems that take a different approach to the one given here. Usually the approach taken is application specific. For instance, FeynCalc[230, 231], Dill[208], HIP[168], and Tracer[174] all perform calculations in "Dirac-algebra-like" settings; but incorporating, say, angular momentum into such systems is somewhat awkward. We desire an elegant extensible system whereby we can uniformly combine approaches.

It should be mentioned that a *Mathematica* package called NCAlgebra[160, 250] attempts to give a set of rules for the handling of non-commutative objects. This appears to be the only widely targeted *Mathematica* package for non-commutative algebra. Indeed, in a wider setting, there are other non-commutative systems of note [9, 63, 190, 201, 228, 309]. Some of these systems are based on non-commutative Gröbner bases[187, 246, 247, 313] and other methodologies. Possibly at a later date, some form of a non-commutative Gröbner basis algorithm could be incorporated into the core of *Mathematica*. However at our stage, we are more interested in being able to specify the inter-relations between hierarchies of objects. It appears that often in our calculations, we are not so interested in finding general non-commutative relations since almost always we know what the governing relations are at the start. For instance, creation and annihilation operators are governed by a Weyl algebra, etc. Thus, using the full generality of non-commutative Gröbner bases would not appear to be worthwhile due to the execution overheads entailed by such an approach. Moreover, the consideration of tensorial non-commutative objects raises several issues, since non-commutative Gröbner bases work with a fixed monomial term ordering (at least in the usual settings[246, 247]. Thus, we encounter the relabeling problem when dealing with tensorially non-commutative objects. Therefore, overall we adopt the approach taken in this chapter.

In relation to some of the other systems available, I believe it is not sufficient to just say something is commutative or it is non-commutative. There are finer degrees of granularity which we must be aware of and able to handle. Indeed, we will be able to handle them. As should be evident after reading this chapter, the method advocated herein is extremely elegant.

The first section, §5.2 *Generic Prototypes*, presents the basic generic structures upon which we will base many of our later structures. Next, in §5.3 *Physics Structures*, we present some

examples using these basic structures and demonstrate how our inheritance paradigm is being used in order to facilitate the expression of these structures. In §5.4 *Example: The Harmonic Oscillator*, we present the harmonic oscillator using the machinery we have so far developed. This includes some work on creation and annihilation operators, normal ordering, matrix elements, propagators, abstract summations, and time independent perturbation theory. Following that, in §5.5 *Example: Angular Momentum*, we illustrate topics in angular momentum, such as commutation relations, raising and lowering operators, spherical harmonics and their commutation relations, representations, couplings and spherical tensor operators.

It should be noted that throughout this application chapter, we do not elaborate on much of the physics. A quick reference to the particular subject under discussion may be given but beyond that it is expected that the reader have the appropriate background in quantum mechanics. The physics being tackled is not sophisticated by any stretch, yet neither is it trivial. Hopefully it is a high enough level to distinctively illustrate the various concepts which we progressively develop, yet at the same time simple enough that anyone with a reasonable knowledge of quantum mechanics should not be overly challenged by the subject matter.

In tackling a given problem in our system, it is important that we can handle *all* attendant calculations in our system, and in a uniform and intuitive way. For instance, someone might say "This part is easy, and one can do it by hand". This could well be true, but almost assuredly the calculations will resurface at some stage when the problem will not be easily doable by hand. For example, the early creation and annihilation operation calculations which we present in §5.4.4 *Creation and Annihilation Operators* can be done by hand. However, once we complicate matters somewhat and transform these operations to quasi-spin systems, the algebra becomes quite problematic by hand. So much so, that it took a doctoral dissertation to resolve some of these issues — see Savage[284]. But yet, our formulation of quasi-spin in our working paradigm differs little from the formulation given in §5.4.4 *Creation and Annihilation Operators*.

Another example is in the canonicalization of tensors — see §6 *An Algorithm for Tensor Simplification*. The need for computer algebra in these calculations has long been recognized [15, 52, 60, 83, 120, 164, 171, 172, 180, 188, 193, 210, 266, 298]. It is quite important that we handle all the operations in a consistent and uniform manner. It is easy for a human to simplify expressions involving one or two Riemann tensors, but beyond that, it is far easier in a computer algebra system with an appropriate package. Thus we see that often one of the hardest parts in formulating a calculation in a symbolic computation system is doing it correctly *once*. After this, working with more complex calculations is essentially just more of the same (if we have designed our algorithms properly), and the computer can usually handle the problem as such. Thus in summary, it is important that we can handle *all* the attendant calculations in our system, and in a uniform and intuitive way.

Let us therefore proceed to develop generic prototypes that embody the properties we need to "do" quantum mechanics.

5.1.2 Initializations

In this chapter we will make extensive use of the *Notation* package to allow the natural input and output of many objects. Let us load up our common set of notations in order that we may freely use notations like ≡, ≢, $(\ldots)_{\text{hp}}$, $\prec_{lex}$, etc. These common notations are documented in §A.1 *Common Notations*.

```
In[1]:= << CommonNotations`
```

Since this chapter is fundamentally based on the language modifications of the previous chapter, we will also load the *Assign* package.

```
In[2]:= << Assign`
```

In addition to the common notations and the notations given in the *Assign* package, there are some additional notations defined in the *Prototypes* package. These notations allow us to easily express dynamic assignment rules with certain adornments. These notations will be discussed when they are needed.

```
In[3]:= << Prototypes`
```

Most of the major underlying structures and functions being referred to are defined in the *Prototypes* package. In some code instances throughout the earlier parts of this chapter, we use relatively simplistic underlying prototypes for illustration. Consequently, we sometimes clear the symbols of the objects we are working with, in order to avoid confusing the issues under discussion. Henceforth in this chapter, various examples will be prefixed by a `Clear` statement.

In the next section we start by presenting the generic prototypes themselves.

5.2 Generic Prototypes

5.2.1 Generic Multilinearity

Let us start by describing and specifying the property of multilinearity. A typical example of a multilinear operator / operation is our non-commutative product previously used in §2.7.4 *Example Calculations from Physics* and §4.6.5 *Dynamic Assignment Example: Non-commutative Expansion*. Another example of a multilinear operation might be the generalized cross product defined as follows.

$$\text{Cross}[a_1, a_2, \ldots, a_n] = a_1 \times a_2 \times \ldots \times a_{n-1} \times a_n = a_1 \times (a_2 \times \ldots \times (a_{n-1} \times a_n) \ldots) \tag{5.2.a}$$

The commutator [A, B]_ = A · B - B · A provides yet another example. That is, the commutator is linear in both its arguments.

These "multilinear" operators are characterized by the following properties. Given a set of constants, C, and two operations, $\oplus$ and $\otimes$, then for all $c \in C$ and for all a_i in our admissible base set of operands, the following should hold.

$$\begin{aligned} &a_1 \otimes a_2 \otimes \ldots \otimes c\, a_m \otimes \ldots \otimes a_n = c * a_1 \otimes a_2 \otimes \ldots a_m \otimes \ldots \otimes a_n \\ &a_1 \otimes a_2 \otimes \ldots \otimes (a_m \oplus a'_m) \otimes \ldots \otimes a_n = \\ &\quad a_1 \otimes a_2 \otimes \ldots \otimes a_m \otimes \ldots \otimes a_n \oplus a_1 \otimes a_2 \otimes \ldots \otimes a'_m \otimes \ldots \otimes a_n \end{aligned} \tag{5.2.b}$$

We need to embody the generic behavior of multilinearity into a set of rules. We will then be able to inherit these rules to specific instances of multilinear operators. Let us denote our prototypical generic multilinear operation by *Generic*$\mathtt{ML}_{\mathtt{Op}}$. (The ML stands for 'multilinear'). We first symbolize all generic operations, environments, and rule sets.

```
In[1]:= Symbolize[a_Op]
        Symbolize[a_Env]
        Symbolize[a_Rules]
```

Technical Note: For consistency, we will always use Rules in the plural, even when the specific set under consideration has a single rule.

Now, any 0 occurring within our multilinear operation in any place makes the whole result 0. (This is a trivial consequence of its multilinearity.)

```
In[4]:= MLZero_Rules = { GenericML_Op[___, 0, ___]_hp :> 0};
```

Besides 0, any "constants" should be factored out of our expressions. This should happen under simplification, expansion, and factorization. Here is a simple set of rules that achieves this goal.

```
In[5]:= MLConstants_Rules = {
          Simplify ↗ (GenericML_Op[ℓ___, c_?GenericConstQ m_., r___] /; c ≠ 1)_hp :>
                c GenericML_Op[ℓ, m, r],
          Factor ↗ (GenericML_Op[ℓ___, c_?GenericConstQ m_., r___] /; c ≠ 1)_hp :>
                c GenericML_Op[ℓ, m, r],
          Expand ↗ (GenericML_Op[ℓ___, c_?GenericConstQ m_., r___] /; c ≠ 1)_hp :>
                c GenericML_Op[ℓ, m, r]};
```

Technical Note: The above set of rules is actually fairly inefficient for large scale problems. Yet, because we have set up our inheritance paradigm in a consistent and abstracted way, it is perfectly permissible to later replace the underlying code structure with a more efficient version.

If any expression has a subexpression with a "plus" in it, then we should distribute this expression over the "plus" under expansion.

```
In[6]:= MLDistribution_Rules = {
          Expand ↗ GenericML_Op /: (expr : GenericML_Op[___, _GenericPlus, ___])_hp :↦
            Distribute[expr_u, GenericPlus_u]};
```

Finally, we can put all these operations together into a coherent set of rules for our generic multilinear operator.

```
In[7]:= GenericML_Op_Rules := MLZero_Rules +{} MLConstants_Rules +{} MLDistribution_Rules ;
```

Technical Note: The assignment above uses delayed evaluation. Consequently, this allows us to change any of the underlying rules such as $\texttt{MLZero}_{\texttt{Rules}}$, or $\texttt{MLConstants}_{\texttt{Rules}}$, or $\texttt{MLDistribution}_{\texttt{Rules}}$, and still have the whole paradigm work without top level change. This is one of the reasons for using a prototypical inheritance paradigm; that is, it allows us to change the implementation, with a relatively small amount of effort — see §5.3.5 *Efficient Matching* for further details.

We can now inherit these multilinear operator behaviors to any desired operator. Before giving specific examples of inheriting these operators, it is convenient to build up some additional prototypical structures. The next subsection creates the prototypical rules for linearity and discusses associativity.

5.2.2 Generic Linearity, Associativity, and Flatness

The generic rules for linearity are just specialized versions of the rules for multilinearity (where it is to be understood that we are implicitly referring to linearity with respect to the first argument). Based on our naming of the multilinear operator, let us denote the generic linear operation by $\textit{Generic}\mathbb{L}_{\texttt{Op}}$. Similar to before, a zero appearing in the first position makes the overall expression zero.

```
In[8]:= LZero_Rules = { GenericL_Op[0, ___]_hp → 0 };
```

Any constants appearing in the first position should be factored out.

```
In[9]:= LConstants_Rules = {
          Simplify ↗
            (GenericL_Op[c_?GenericConstQ m_., r___] /; c ≠ 1)_hp :↦ c GenericL_Op[m, r],
          Factor ↗
            (GenericL_Op[c_?GenericConstQ m_., r___] /; c ≠ 1)_hp :↦ c GenericL_Op[m, r],
          Expand ↗
            (GenericL_Op[c_?GenericConstQ m_., r___] /; c ≠ 1)_hp :↦ c GenericL_Op[m, r]};
```

In addition, if the first argument to the generic linear operator is an expression whose head is a generic plus, then distribute over this head.

```
In[10]:= LDistribution_Rules = {
           Expand ↗ GenericL_Op[GenericPlus_hp @ args___, rest___]_hp :↦
                 GenericPlus @@ (GenericL_Op[#, rest] & /@ {args})};
```

Finally, as before, we join this collection of rules together to obtain the prototypical set of rules for a generic linear operator.

```
In[11]:= GenericL_Op_Rules := LZero_Rules +{} LConstants_Rules +{} LDistribution_Rules ;
```

Let us now turn to associativity and thus "flatness". These are standard concepts in *Mathematica*, and traditionally, flat operations are implemented in terms of setting the attributes `Flat` and `OneIdentity`. The role of attributes within our assignment and inheritance paradigm was extensively discussed in §4.7.2 *Attribute Values*. Indeed, it may be insightful to the present discussion to review that subsection.

We have not yet unilaterally declared that our generic operators are associative or "flat", nor should we. Such declarations are only true in selective circumstances. For instance, the multiple cross product, (5.2.a), is neither associative nor flat. To specify that an operator is flat in a consistent manner in our inheritance paradigm, we must create a prototypical set of rules for a generic associative flat operator. If we use attributes, such a set of rules would look like the following.

```
In[12]:= AssociativeViaAttributes_Rules = {
           AttributeIsSetQ[Generic_Op, Flat]_hp → True,
           AttributeIsSetQ[Generic_Op, OneIdentity]_hp → True};
```

However, we should probably avoid using the *Mathematica* attributes `Flat` and `OneIdentity`. Although they appear at first to be able to describe associative operations, their implementation in *Mathematica* unfortunately makes them slightly crippled. The main reason is that flat heads cannot be removed. For instance, if `f` is flat, then the rule `f[x_]:=x` results in an infinite loop. This makes it very hard to emulate the normal behavior of *Mathematica*'s functions like `Plus` and `Times`, since for both of these functions, `Plus[`*any*`]` → *any* and `Times[`*any*`]` → *any*. Also, there are other problems which are surmountable but hard to avoid. These problems have to do with matching structures when flat. For further discussion of this point, see §B.1 *Attributes, Pattern Matching, and Associativity*.

The other possible way in which we can implement associativity, and indeed the way we will adopt, is via some simple rules.

```
In[13]:= GenericFlat_Rules =
           {Generic_Op /: (Generic_Op[args___] /; Blank[Generic_Op] ∈? {args}_u)_hp :>
                 Flatten[(Generic_Op @ args)_u, ∞, Generic_Op]};
         GenericOneIdentity_Rules = {Generic_Op /:
                 (expr : Blank[Generic_Op] /; expr_ren ≡ 1)_hp :> expr_u [[1]]};

In[15]:= Associative_Rules := GenericFlat_Rules +{} GenericOneIdentity_Rules
```

Again, we delay the presentation of examples using these prototypical rule sets until after the next two subsections.

5.2.3 Generic Constants

Previously, in §2.7.4 *Example Calculations from Physics* and §4.6.5 *Dynamic Assignment Example: Non-commutative Expansion*, we gave a somewhat informal implementation of constants. In keeping with our generic structures, we need to create a prototypical set of rules which embody "constants". In our more formal generic version we require, in the interests of execution speed, that all results are cached. The following set of rules allows the declaration, the un-declaration, and the handling of "constants".

```
In[16]:= GenericConstantQ_Rules = {
           GenericConstQ[args_GenericTimes]_bg :>
             (GenericConstQ @ args = And @@ GenericConstQ /@ Hold @@ args_u),
           GenericConstQ[args_GenericPlus]_bg :>
             (GenericConstQ @ args = And @@ GenericConstQ /@ Hold @@ args_u),
           GenericConstQ[arg_^num_-]_bg :>
             (GenericConstQ @ arg^num = GenericConstQ[arg_u] /\ GenericConstQ[num_u]),
           GenericConstQ[arg_?NumericQ]_bg :> True,
           GenericConstQ[other_]_bg :> False,

           DeclareGenericConstant[expr_]_bg :>
             (DownValues @ GenericConstQ = Select[DownValues @ GenericConstQ,
                 FreeQ[#, expr, {0, ∞}] &];
               GenericConstQ @ expr = True;),

           DeclareGenericConstant[exprs : {__}]_bg :>
             (DownValues @ GenericConstQ = Select[DownValues @ GenericConstQ,
                 FreeQ[#, Alternatives @@ exprs, {0, ∞}] &];
               (GenericConstQ @ expr = True) &_expr /@ exprs;),

           DeclareGenericUnconstant[expr_]_bg :>
             (DownValues @ GenericConstQ = Select[DownValues @ GenericConstQ,
                 FreeQ[#, expr, {0, ∞}] &];
               GenericConstQ @ expr = False;),

           DeclareGenericUnconstant[exprs : {__}]_bg :>
             (DownValues @ GenericConstQ = Select[DownValues @ GenericConstQ,
                 FreeQ[#, Alternatives @@ exprs, {0, ∞}] &];
               (GenericConstQ @ expr = False) &_expr /@ exprs;)};
```

Why in this case did we create a generic set of definitions for what constitutes a constant? It turns out that it is convenient to sometimes have more than one class of "constants". As in any object-oriented language, there is always the issue of what to encapsulate as a class, or equivalently where to "draw boundaries". In the end, it is a matter of style. In some object-oriented languages, everything is an object; in others, mixtures of objects/non-objects are allowed. For a discussion of these concepts, see [18]. In our language model, we allow a mixture of object-oriented programming and "straight" programming. This allows us to create instances without classes.

Technical Note: For classes of "constants" other than the one considered in this subsection, see §5.4.7 *Aside: Hermitian Conjugates* or §5.5.1 *Primitive Cartesian Angular Momenta*.

We can recreate a cached version of our previous simple `ConstQ` function by simply inheriting this structure.

```
In[17]:= Assign[GenericConstantQ_Rules /. {
                  GenericConstQ → ConstQ,
                  GenericTimes → Times,
                  GenericPlus → Plus,
                  DeclareGenericConstant → DeclareConstant,
                  DeclareGenericUnconstant → DeclareUnconstant}]
```

Here are some constant declarations.

```
In[18]:= DeclareConstant[{m, ħ, j}]
```

The following expression can be considered constant since its constituents are constant.

```
In[19]:= ConstQ[√(m j (j + 1))]
Out[19]= True
```

In contrast, since `p` has not been declared as a constant, the following expression is not considered constant.

```
In[20]:= ConstQ[p + 1]
Out[20]= False
```

On a related note, it would be desirable if *Mathematica* had a more general facility for handling assumptions. As it stands, assumptions can be used in the simplify functions, such as `Simplify` and `FullSimplify`, as well as the integrate functions, such as `Integrate` and `FourierTransform`, etc.— although there is a different calling syntax in these two function classes[343]. In §5.4 *Example: The Harmonic Oscillator*, we will see that it would be desirable to uniformly assert some assumptions throughout an entire calculation. Maple has some degree of assumption handling, but apparently it is somewhat problematic. I am led to believe efforts are continuing at Wolfram Research to introduce a more general assumptions mechanism [private communication - Adam Strzebonski].

Before we utilize the generic prototypes for multilinearity or linearity, let us introduce one last prototype — that of generic differentiation.

5.2.4 Generic Differentiation

Differentiation is an operation that arises in any setting involving calculus. For instance, our partial and covariant derivatives are specific cases of this kind of generic operation — see §7.2.5 *Partial and Covariant Derivatives*. Such structures will arise in many situations, and thus it is necessary for us to create a set of rules to embody the generic properties of differentation.

We first need a notation for generic partial derivatives. Because the *Mathematica* front end is so aggressive with the grouping of any expression involving '∂', we need to use a grid box with zero column spacing to juxtapose the symbol *Generic* and the operator ∂. That way, when we subscript the combined construct, brackets are not added around the structure. (Unfortunately, tag boxes are not sufficient in this regard. The interested reader can examine the underlying structure in the usual way.)

```
In[21]:= Symbolize[Generic∂]

In[22]:= Notation[Generic∂_var_ arg_ ⟺ GenericD[arg_, var_]]
```

The notation for generic partial derivatives is now functioning as desired.

```
In[23]:= Generic∂_t f@π@t + x

Out[23]= x + Generic∂_t f[π[t]]

In[24]:= FullForm @ %

Out[24]//FullForm=
Plus[x, GenericD[f[Pi[t]], t]]
```

We want generic partial derivatives to be linear in their first argument. Moreover, we want this linearity to always be active. Thus we include rules almost exactly like those in `LConstantsRules` above. In addition, we must add the specific behaviors of generic differentiation when it is active.

```
In[25]:= GenericDProductRules = {GenericEnv ↗ (Generic∂var_ arg___GenericTimes)hg :>
            GenericPlus @@ MapEach[Generic∂var #1 &, arg] /; argLength > 1};

In[26]:= GenericDPowerRules =
          {GenericEnv ↗ (Generic∂var_ a_^n_Integer?Positive)hg :> With[{d = Generic∂var a},
             GenericPlus @@ ({GenericTimes[d, a^(n-1)]} +() Table[GenericTimes[a^i, d, a^(n-i-1)],
                {i, 1, n - 2}] +() {GenericTimes[a^(n-1), d]})]};

In[27]:= GenericDRules = {
            (Generic∂r___ (c_?GenericConstQ m_.) /; c ≠ 1)hg :> c (Generic∂r m),
            (Generic∂rest___ GenericPlushg[args___])hg :>
             GenericPlus @@ (Generic∂rest # &) /@ {args},
            (Generic∂var_ 1)hg :> 0} +()
           GenericDProductRules +()
           GenericDPowerRules ;
```

The rules in $GenericD_{Rules}$ depend on the routine `MapEach`. Given a function, say f, and an expression, say *expr*, `MapEach` creates n duplicates of *expr* with f mapped onto the i^{th} element of the i^{th} duplicate of *expr*.

```
In[28]:= MapEach[f_, expr : (_ @ __)] := Table[MapAt[f, expr, i], {i, 1, exprLength}]
```

We are now in a position to start developing specific structures with these generic operations. In the next section, the first such structure that we will add is that of non-commutative multiplication.

5.3 Physics Structures

5.3.1 Non-Commutative Times I

In previous subsections of this thesis, we have used the operation `NonCommutativeTimes` several times. We have used this operation to embody a somewhat generic non-commutative times operation. This operation has been polymorphically used over different data types. Recall that it was used when an operator acted on a ket in §2.7.4 *Example Calculations from Physics,* for instance $\hat{J}_+ \cdot \hat{J}_- \cdot \left| j_{\hat{J}}, m_{\hat{J}_z} \right\rangle$. It was also used between a differentation operator and a vector in §3.5.5 *Simple Cartesian Vector Calculus I,* for instance,

$$\hat{\partial}_a \cdot \hat{\partial}_b \cdot \vec{\mathbf{v}}_d \, \epsilon_{cbd} \, \epsilon_{iac}$$

Yet another example of its usage was as a general "non-commutative" operation in §4.6.5 *Dynamic Assignment Example: Non-commutative Expansion,* for instance `a·(b+2c)`.

We would like our operation to be general, yet if needed, we want to be able to further specialize it through our inheritance paradigm. As before, we use the · operation for our non-commutative times.

```
In[1]:= InfixNotation[·, NonCommutativeTimes]
```

It is also highly convenient to have a *"mono-fix"* form for our non-commutative times operation. We choose the following form.

Technical Note: Although it is not a standard concept, we use the term "*mono-fix*" to refer to the notation it intuitively conjures up. For instance, assume as is standardly the case, that the infix expression `a+b` translates/parses to `Plus[a,b]`, and `a+b+c` translates to `Plus[a,b,c]`, etc. What then, is the pre-image or pre-translation of `Plus[a]`? Is it `+a`? For our usage, this is the *mono-fix* form.

```
In[2]:= Notation[expr_nc ⟺ NonCommutativeTimes[expr_]]
```

Since our non-commutative times operation is a multilinear operation, we inherit the rules of multilinearity. We make the necessary replacements of `NonCommutativeTimes` for *Generic*$\mathtt{ML}_{\mathtt{Op}}$ and `Plus` for *Generic*`Plus` since we do not have any variants of plus that we need to distinguish between.

```
In[3]:= Assign[ GenericML_OpRules /. {
                  GenericML_Op → NonCommutativeTimes,
```

```
    GenericTimes → NonCommutativeTimes,
    GenericPlus → Plus,
    GenericConstQ → ConstQ}]
```

We need our non-commutative times operation to be both flat and associative.

```
In[4]:= Assign [Associative_Rules /. {Generic_Op → NonCommutativeTimes}]
```

Finally, powers distribute over `Plus`, but only under expansion.

```
In[5]:= Expand ↗ (ℓ___ · sum_Plus^(n_Integer?Positive) · r___)  := 
          Plus @@ (ℓ · ## · r &) @@@
                Distribute[ Table[ List @@ sum, {n}], List]
```

Our non-commutative times is now functioning in a capacity as a non-commutative operator.

```
In[6]:= a · (b + c)^3

Out[6]= a · (b + c)^3

In[7]:= Expand @ %

Out[7]= a · b · b + a · b · b · c + a · b · c · b + a · b · c · c +
          a · c · b · b + a · c · b · c + a · c · c · b + a · c · c · c

In[8]:= x · (-2 v z) · (k - 1/3)^2

Out[8]= x · (-2 v z) · (k - 1/3)^2

In[9]:= Expand @ %

Out[9]= -2 x · (v z) · k · k + 2/3 x · (v z) · k · 1 + 2/3 x · (v z) · 1 · k - 2/9 x · (v z) · 1 · 1

In[10]:= DeclareConstant[v]

In[11]:= Expand @ %%

Out[11]= -2 v x · z · k · k + 2/3 v x · z · k · 1 + 2/3 v x · z · 1 · k - 2/9 v x · z · 1 · 1
```

In doing non-commutative calculations, it often arises that expanding seemingly small expressions can give rise to an alarming number of terms. For instance, consider the following relatively simple non-commutative product.

```
In[12]:= a · (b + c)^6 · (d + e)^6

Out[12]= a · (b + c)^6 · (d + e)^6
```

Expanding this product leads to a large number of terms.

```
In[13]:= Length @ Expand @ (a · (b + c)^6 · (d + e)^6)

Out[13]= 4096
```

Comparatively, the corresponding commutative product has far fewer terms.

```
In[14]:= Length @ Expand[a (b + c)^6 (d + e)^6]

Out[14]= 49
```

We have yet to create an additional behavior for powers. We will soon do this, but first it is illuminating to give a few further examples using our generic prototypical structures.

5.3.2 Commutators

In this subsection we present several further simple examples that use our generic prototypical structures. The first of these is the *commutator*. This structure is ubiquitous in physics and is seen in areas too numerous to list. Its main use is probably in quantum mechanics. It interrelates two non-commuting observables or operators, for example, X and P, or L_x and L_y — see Shankar[292], Cohen-Tannoudji[68], etc. It is also used in the basic quantization scheme of quantum mechanics, where the quantized versions of the basic variables must satisfy the canonical commutation relations — see [68, 292]. Indeed, a similar approach is taken in quantum electrodynamics, and the canonical quantization scheme of quantum field theory [69, 173, 186]. In short, it is an extremely widely used concept. Let us therefore define the commutator of two objects.

How do we implement such a concept? As usual, we start by defining a notation for our object, in this case a commutator.

```
In[15]:= Notation[[x_, y_]_ ⟺ Commutator[x_, y_]]
```

The commutator is a multilinear object, so we force it to inherit the rules of multilinearity.

```
In[16]:= Clear @ Commutator;
         Assign [ GenericML_OpRules /. {
                    GenericML_Op → Commutator,
                    GenericTimes → Times,
                    GenericPlus → Plus,
                    GenericConstQ → ConstQ} ]
```

To complete the definition of the commutator, we need only add the single rule of how a commutator is expanded, along with the simplification that anything commutated with itself is zero.

```
In[18]:= Expand ↗ [x_, y_]_ := x · y - y · x
         [x_, x_]_ := 0
```

Commutators now have all the factoring, simplification, and expansion properties one would expect of them.

```
In[20]:= DeclareConstant[j]

In[21]:= [2 j a, b - c]_

Out[21]= [2 a j, b - c]_
```

Commutators simplify as desired.

```
In[22]:= Simplify @ %

Out[22]= 2 j [a, b - c]_
```

Commutators are also expanded as one desires.

```
In[23]:= Expand @ %

Out[23]= 2 j a · b - 2 j a · c - 2 j b · a + 2 j c · a
```

Self commutation is automatically simplified to zero.

```
In[24]:= 2 j [a, a]_

Out[24]= 0
```

For illustration purposes, suppose we only want expansion to expand over sums in our commutator, and instead leave to some other environment, say `Act`, the explicit re-expression of the commutator itself in terms of non-commutative products. This is easily achieved by removing the expansion rule for commutators under the environment `Expand`.

```
In[25]:= Expand ↗ [x_, y_]_ =.
```

Now expansion will not, by default, expand commutators completely, but only to the limits given by our generic multilinear operators.

```
In[26]:= Expand @ [2 j a, b - c]_

Out[26]= 2 j [a, b]_ - 2 j [a, c]_
```

We may want to do this in order to perform computations entirely within a commutator formalism — for instance, see [251]. We could easily introduce the "complete expansion" rule in another environment, say `Act`.

```
In[27]:= Act ↗ ([x_, y_]_)_□ := x · y - y · x
         Act @ any_ := DynamicBar @ any
```

In the last definition involving `Act`, we enforced the fact that the dynamic rules of `Act` do not extend outside their context. This was explained in §4.7.6 *Caveats about Dynamic Rules I: Renegade Environmental Rules*. In addition, it is nice for the `Act` environment to have the same linear expansion properties as the environment `Expand`. Thus, we again simply force `Act` to inherit the rules for a multilinear operator.

```
In[29]:= Assign [ GenericML_{Op_Rules} /. {
                    GenericML_{Op} → NonCommutativeTimes,
                    GenericTimes → NonCommutativeTimes,
                    GenericPlus → Plus,
                    GenericConstQ → ConstQ,
                    Expand → Act} ]
```

5.3.3 Continued Commutators

Related to the commutator, defined in the previous subsection, is the *continued commutator*. A continued commutator is just a nested commutator of a given depth of the form $[s, [s, \ldots [s, [s, h]] \ldots]]$ for n nestings. For example, here is a 6-nested continued commutator.

```
In[30]:= [s, [s, [s, [s, [s, [s, h]_]_]_]_]_]_

Out[30]= [s, [s, [s, [s, [s, [s, h]_]_]_]_]_]_
```

We can expand this nested commutator by acting on it with our commutator expansion rules given in the previous subsection.

```
In[31]:= Act @ %

Out[31]= h·s·s·s·s·s·s - 6 s·h·s·s·s·s·s +
         15 s·s·h·s·s·s·s - 20 s·s·s·h·s·s·s +
         15 s·s·s·s·h·s·s - 6 s·s·s·s·s·h·s + s·s·s·s·s·s·h
```

With a minor amount of insight, the coefficients in the above expansion can easily be seen to be the binomial coefficients, up to an alternating sign. Also, the above commutator is linear in its "main" argument, which in this case is h. Let us verify this.

```
In[32]:= Expand /@ Act[
           [s, [s, [s, [s, [s, [s, h0 + 2 h1]_]_]_]_]_]_ ==
             [s, [s, [s, [s, [s, [s, h0]_]_]_]_]_]_ +
               2 [s, [s, [s, [s, [s, [s, h1]_]_]_]_]_]_ ]

Out[32]= True
```

Let us consequently introduce `ContinuedCommutator` as a linear operation. This is accomplished by inheriting the rules for linearity. As always, using inheritance affords a degree of elegance and structure, and is also beneficial for efficiency reasons.

```
In[33]:= Assign [ GenericLOpRules /. {
                    GenericLOp → ContinuedCommutator,
                    GenericTimes → NonCommutativeTimes,
                    GenericPlus → Plus,
                    GenericConstQ → ConstQ} ]
```

Without any further rules, our continued commutators are by default inert.

```
In[34]:= ContinuedCommutator[2 t + h, s, 6]

Out[34]= ContinuedCommutator[h + 2 t, s, 6]
```

Yet, due to the inherited generic rules, our continued commutators are linear in their first argument. Observe this behavior in the following.

```
In[35]:= Expand @ %
```

```
Out[35]= ContinuedCommutator[h, s, 6] + 2 ContinuedCommutator[t, s, 6]
```

Let us now add the "binomial-like" expansion rule demonstrated above.

```
In[36]:= Act ↗ ContinuedCommutator[h_, s_, m_Integer?NonNegative]_hy := 
           Plus @@
             Table[(-1)^(n+1) (-1)^(m+1) Binomial[m, n] (NonCommutativeTimes @@ Join[
                                 Table[s, {n}], {h}, Table[s, {m-n}]]), {n, 0, m}];
```

Expansions of our continued commutators now occur according to this rule.

```
In[37]:= Act @ ContinuedCommutator[h, t, 5]

Out[37]= -(h·t·t·t·t·t) + 5 t·h·t·t·t·t - 10 t·t·h·t·t·t +
           10 t·t·t·h·t·t - 5 t·t·t·t·h·t + t·t·t·t·t·h
```

This, of course, agrees with the result of just expanding out the nested commutators.

```
In[38]:= % == [t, [t, [t, [t, [t, h]_]_]_]_]_ // Act

Out[38]= True
```

5.3.4 Baker-Campbell-Hausdorff Expansions

Some applications in operator theory and quantum mechanics, and in particular, quantum electrodynamics and quantum field theory, make use of the Baker-Campbell-Hausdorff expansion [23, 135, 186]. This expansion expresses the product $e^{i\hat{S}}\,\hat{H}\,e^{-i\hat{S}}$ as a power series of continued commutators of $\hat{H}$ by $\hat{S}$. Formally it can be expressed as the following.

$$\begin{aligned} e^{i\hat{S}}\,\hat{H}\,e^{-i\hat{S}} &= \left(1+\tfrac{i\hat{S}}{1!}+\tfrac{i^2\hat{S}^2}{2!}+\ldots\right)\hat{H}\left(1+\tfrac{(-i)\hat{S}}{1!}+\tfrac{(-i)^2\hat{S}^2}{2!}+\ldots\right) \\ &= \hat{H}+i\left[\hat{S},\hat{H}\right]_- + \tfrac{i^2}{2!}\left[\hat{S},\left[\hat{S},\hat{H}\right]_-\right]_- + \ldots + \\ &\quad \tfrac{i^n}{n!}\left[\hat{S},\left[\hat{S},\ldots\left[\hat{S},\hat{H}\right]_-\ldots\right]_-\right]_- + \ldots \end{aligned} \tag{5.3.a}$$

We denote a Baker-Campbell-Hausdorff expansion by the function/data-structure `BCHExpansion`. Note that the Baker-Campbell-Hausdorff expansion is linear in its main argument. This can easily be seen from (5.3.a). For efficiency, let us consequently inherit the generic linearity rules.

```
In[39]:= Assign [ GenericL_Op_Rules /. {
                    GenericL_Op → BCHExpansion,
                    GenericTimes → NonCommutativeTimes,
                    GenericPlus → Plus,
                    GenericConstQ → ConstQ} ]
```

Technical Note: We do not include a notation for `BCHExpansion` objects since there is no conventional notation which encapsulates these objects other than expressions like $e^{i\hat{S}}\,\hat{H}\,e^{-i\hat{S}}$. Possible notations that come to mind are $\left[\hat{S},\hat{H}\right]_{-n}$, but this raises the dilemma of wondering if the expansion occurs like $\left[\hat{S},\left[\hat{S},\ldots\left[\hat{S},\hat{H}\right]_-\ldots\right]_-\right]_-$ or like $\left[\left[\left[\hat{H},\hat{S}\right]_-,\hat{S}\right]_-,\ldots,\hat{S}\right]_-$. The former corresponds to $e^{i\hat{S}}\,\hat{H}\,e^{-i\hat{S}}$ while the latter corresponds to $e^{-i\hat{S}}\,\hat{H}\,e^{i\hat{S}}$.

As before, we could demonstrate that BCHExpansion objects have the expansion and simplification behavior associated with a linear operator.

```
In[40]:= Expand @ BCHExpansion[2 h + x, s, 4]

Out[40]= 2 BCHExpansion[h, s, 4] + BCHExpansion[x, s, 4]
```

Let us now add the single rule for the expansion (5.3.a) under the Act environment.

```
In[41]:= Act ↗ BCHExpansion[h_, s_, n_Integer?NonNegative] :=
           Module[{j}, Sum_{j=0}^{n} (i)^j / j! ContinuedCommutator[h, s, j]]
```

Since continued commutators are fully expanded under the Act environment, our BCHExpansion objects will get expanded to continued commutators and then further expanded to non-commutative products.

```
In[42]:= Expand @ Act @ BCHExpansion[h + x, s, 3]

Out[42]= h + x - i h·s + i s·h + i s·x - i x·s - h·s·s/2 + s·h·s -
         s·s·h/2 - s·s·x/2 + s·x·s - x·s·s/2 + 1/6 i h·s·s·s -
         1/2 i s·h·s·s + 1/2 i s·s·h·s - 1/6 i s·s·s·h -
         1/6 i s·s·s·x + 1/2 i s·s·x·s - 1/2 i s·x·s·s + 1/6 i x·s·s·s
```

We can easily verify that our rule for BCHExpansion expansion agrees with the formal Baker-Campbell-Hausdorff expansion (5.3.a).

```
In[43]:= Expand @ Act[(h + x) + i [s, h + x]_ +
                      i^2/2! [s, [s, h + x]_]_ + i^3/3! [s, [s, [s, h + x]_]_]_] == %

Out[43]= True
```

With the simple way we have stated the above rule, it is important to realize that we have achieved a good degree of efficiency in a clean and consistent manner. Here are the two timings for the equivalent yet different ways to obtain the same result.

```
In[68]:= expr1 = Act @ ContinuedCommutator[h + x, s, 60]; // Timing

Out[68]= {1.23333 Second, Null}

In[69]:= expr2 = Act @ Nest[[s, #]_ &, h + x, 60]; // Timing

Out[69]= {24.3667 Second, Null}

In[70]:= Expand @ expr1 == Expand @ expr2

Out[70]= True
```

One might think that this result is trivially obvious, that is, we just have a better way to reduce continued commutators. However, the important thing to realize about the overall example is

that we specified the rule just once in the Act environment, yet all the other associated behavior is occurring as we would desire.

We indulge for a moment in a piece of informal comment. Hopefully, in progressing through this section, the reader just "reads" each line and thinks "yes, that is fairly obvious" or "yes, that seems reasonable". If we have achieved this, then we have accomplished our goals. We have managed to state in a clear, consistent, extensible, and uniform way, a simple translation of the way we would work on paper. All of the functions like Expand, Simplify, etc., work consistently with the objects just defined. This was made possible only through our generic linear structures and dynamic rules, both of which are parts of our whole inheritance paradigm. Thus, our language modifications are instrumental in allowing the elegant expression of mathematical properties.

To close this subsection, it should be pointed out that in the last three subsections, we split off the full expansion rules for our various objects into the environment Act, for the sole reason of illustration. Thus, let us merge the rules presently defined in Act back into Expand. As usual, we can perform this with our inheritance paradigm.

```
In[44]:= Assign [ EnvironmentValues @ Act /. {Act → Expand}]
```

Expand now fulfills both its normal expansion role, as well as expanding out the special structures introduced so far.

```
In[45]:= Expand @ BCHExpansion[h + x, s, 2]
```

$$\text{Out[45]= } h + x - i\,h \cdot s + i\,s \cdot h + i\,s \cdot x - i\,x \cdot s - \frac{h \cdot s \cdot s}{2} + s \cdot h \cdot s - \frac{s \cdot s \cdot h}{2} - \frac{s \cdot s \cdot x}{2} + s \cdot x \cdot s - \frac{x \cdot s \cdot s}{2}$$

Technical Note: We did not assign all of the rules of Act. In particular, we did not assign the rule Act[*any*] :↦ DynamicBar[*any*], for to have done so would have circumvented the expansion carried out by the "system". This brings up the question of when one should inherit from one structure to another. Is it "safe" to inherit from just "any old structure"? In our extremely flexible environment, in the end, it is a matter of style.

In summary, by the simple inheritance of linearity and the addition of a single rule, our BCHExpansion objects are acting in a manner consistent with a full and faithful implementation of Baker-Campbell-Hausdorff expansions.

5.3.5 Efficient Matching

In the foregoing material, we have presented several generic prototypes for operations that we are interested in. We have given several simple examples of inheriting these structures and the consequent usage of the instantiated systems. There are several key reasons why we have developed the generic prototype paradigm in the manner that we have: (*i*) it gives structure to our overall calculations as has hopefully been witnessed; (*ii*) it allows the reuse of our code structures; and, (*iii*) by having a layer of abstraction between the specification and the implementation, we can change the implementation layer with relative impunity. In fact, these reasons are generally true of abstract datatypes/object-oriented programming[18, 38, 125, 234, 235]. It is this last stated reason, that is reason (*iii*), which will occupy our attention in this section.

It was pointed out in §5.2.1 *Generic Multilinearity* that the rules we used for linearity, $Generic\mathtt{ML}_{\mathtt{Op}_{\mathtt{Rules}}}$, were not as fast or as efficient as they could be. For reference, the rules we are referring to are those that were instantiated from the following generic form.

$$Generic_{\mathtt{Env}} \nearrow (Generic\mathtt{ML}_{\mathtt{Op}}[\ell\_\_\_, c\_?Generic\mathtt{ConstQ}\, m\_.., r\_\_\_] /; c \neq 1)_{\mathtt{hp}} :\mapsto c\, Generic\mathtt{ML}_{\mathtt{Op}}[\ell, m, r] \tag{5.3.b}$$

Although the intent of (5.3.b) is relatively transparent, it is unfortunately slow compared to other equivalent rules. The following rule appears quite horrible, but compared to (5.3.b), it can typically be around 50 times faster for larger expressions. However, in all fairness, (5.3.b) is equally efficient for smaller expressions.

```
In[46]:= OptimizedMLConstants_Rules = {Generic_Env ↗ GenericML_Op[args___]_hp :↦
            Block[{theSplit = Flatten /@ Transpose @ Cases[{args},
                  a_?GenericConstQ | a_?GenericConstQ b_ | b_ → {{a}, {b}}]},
              (Times @@ theSplit[[1]]) (GenericTimes @@ theSplit[[2]]) /; First @ theSplit ≠ {}]};
```

Technical Note: Due to an already long thesis, I have not provided details to justify the claim that it is more than 50 times faster. However, it is a relatively simple matter to do so.

To incorporate our optimized rule into our generic prototypes, we simply change the $\mathtt{MLConstants}_{\mathtt{Rules}}$ and re-inherit the structures we would like to change.

```
In[47]:= MLConstants_Rules =
           (OptimizedMLConstants_Rules /. Generic_Env → Expand) +{}
             (OptimizedMLConstants_Rules /. Generic_Env → Simplify) +{}
             (OptimizedMLConstants_Rules /. Generic_Env → Factor);
```

Let us now clear the old definitions for `NonCommutativeTimes` and inherit the new prototypes afresh.

```
In[48]:= Clear @ NonCommutativeTimes
```

```
In[49]:= Assign [ GenericML_OpRules /. {
                    GenericML_Op → NonCommutativeTimes,
                    GenericTimes → NonCommutativeTimes,
                    GenericPlus → Plus,
                    GenericConstQ → ConstQ}]
```

Again, we need our non-commutative times operation to be both flat and associative.

```
In[50]:= Assign [Associative_Rules /. {Generic_Op → NonCommutativeTimes}]
```

We again add the single rule for distributing powers over `Plus`, but only under expansion.

```
In[51]:= Expand ↗
          (f____ · sum_Plus^n_Integer?Positive · r____)_tag := Plus @@ (f · ## · r &) @@@
           Distribute[ Table[ List @@ sum, {n}], List]
```

This completes the update of the rules specifying the behavior of our non-commutative times operation. Let us demonstrate that simple calculations still proceed in the same manner as they did before.

```
In[52]:= x · (-2 v z) · (k - 1/3)^2

Out[52]= x · (-2 v z) · (k - 1/3)^2

In[53]:= Expand @ %

Out[53]= -2 v x · z · k · k + 2/3 v x · z · k · 1 + 2/3 v x · z · 1 · k - 2/9 v x · z · 1 · 1

In[54]:= Expand [v · a^2]

Out[54]= a^2 v
```

Technical Note: We have not updated the generic prototypes for linearity since they would be no faster than using the new "optimized rules". This transpires since the "optimized rules" were optimized for multiple arguments and the single linearity rule has only a single argument.

Since we have created a layer of abstraction between our implementation and our generic prototypes, we can change the underlying implementation without changing the overall specification. This is one of the strong advantages / benefits of abstract data types / object-oriented programming. Let us next move on, finalizing the non-commutative times operation with respect to the changes and enhancements of the previous subsections.

5.3.6 Non-Commutative Times II

Let us now re-examine our implementation of `NonCommutativeTimes`. There are several deficiencies and flaws with our implementation. Consider the following.

```
In[55]:= DeclareConstant [k]
```

```
In[56]:= Simplify[k · a]

Out[56]= a k
```

Owing to the flat nature of our non-commutative times operation, the non-commutative wrapper surrounding a has been removed entirely. This arises directly from our generic rules for associativity given in §5.2.2 *Generic Linearity, Associativity, and Flatness*. Is this behavior always desired? Having raised the question, let us temporarily postpone the answer. Instead, let us comment on powers. Consider the following.

```
In[57]:= Simplify[a · a · b · b · k · b]

Out[57]= k a · a · b · b · b
```

Upon examining the above result, one might wonder why this has not simplified to $k\,a^2 \cdot b^3$, where the power is a "non-commutative power". Let us introduce rules for such simplification.

```
In[58]:= Simplify ↗ (lhs___ · b_^m_. · b_^n_. · rhs___)_inp := lhs · b^(n+m) · rhs
```

Our powers now work correctly.

```
In[59]:= Simplify[a · a · b · b · k · b]

Out[59]= k a^2 · b^3
```

Yet this clashes with the simplification of `NonCommutativeTimes[any]` ⟶ *any*. For instance, under simplification we have `a·a·a` ⟶ `NonCommutativeTimes[a^3]` ⟶ a^3. So, should we in fact reduce mono-fix non-commutative forms? The answer is yes, provided the head is not `Power`. Actually, let us defer the implementation of structures which consistently add powers to our non-commutative products, since the above examples motivate our next development.

5.3.7 "Associative" Structures and Associativity

In the previous subsection, it was shown that for operations without the `Flat` attribute, we must specify patterns as something like *lhs*___ · *pattern* · *rhs*___. For instance, in that subsection we used the pattern *lhs*___ · b_^m_. · b_^n_. · *rhs*___. This style of pattern is opposed to the more compact form of just the "central" pattern if the `Flat` and `OneIdentity` attributes are set. Yet, using the `Flat` and `OneIdentity` attributes can be problematic for pattern matching, efficiency, and the inheritance of attribute values. These problems are documented in §B.1 *Attributes, Pattern Matching, and Associativity* as well as in §4.7.2 *Attribute Values*. How can we accommodate both succinctness and efficiency? We do this by introducing a new notation that will allow us to "grab" the patterns and internally transform them into a new form. The new notation has the same form for pattern specifications as that used when the operations have `Flat` attributes, yet we avoid the aforementioned problems of using attributes.

Technical Note: In essence, the efficiency problem with flat operators is due to the fact that they "try" awfully hard to find a match to a given pattern by "examining" all possible rearrangements of the expression to be matched.

The *Prototypes* package defines several notations and functions for the automatic transcription of "centrally patterned" rules into their corresponding "complete patterned" rules. This transcription is applicable to any non-flat and "sensible" sort of operation. The notations use an overscripted double-struck 'a' over the corresponding normal operation; that is, $\overset{a}{:\!\!\mapsto}$, $\overset{a}{\rightarrow}$, $\overset{a}{=}$, $\overset{a}{:=}$ (where 'a' is to be suggestive of 'associative'). In essence these notations transform "centrally patterned" rules to newly patterned rules which act in a flat or associative manner. Thus, overall, we refer to these newly introduced notations / structures as "associative" rules or "associative" assignments.

Let us give an example concerned with powers since this topic was eluded to above. Consider the operation ⊗ (which has yet to be defined). Assume, further, that we desire a "power-simplification" law over ⊗ to be active under a specific environment, say `foo`. We could create this as follows.

```
In[60]:= foo ↗ b_^m_. ⊗ b_^n_. :=a= b^(n+m)
```

Now, due to the presence of the "associative" dynamic assignment, that is $\overset{a}{:=}$, the internal pattern is created in a fashion that acts in a flat manner.

```
In[61]:= ?? foo

         Global`foo

         foo has the following environment rules :

         foo ↗ Prototypes`Private`lhsOp____ ⊗ b_^m_. ⊗ b_^n_. ⊗ Prototypes`Private`rhsOp____ :=
           Prototypes`Private`lhsOp ⊗ b^(n+m) ⊗ Prototypes`Private`rhsOp
```

Evidently, the "central patterns" have been translated into the following fuller form.

$$\texttt{foo} \nearrow lhsOp\_\_\_\_ \otimes b\_^{m\_\cdot} \otimes b\_^{n\_\cdot} \otimes rhsOp\_\_\_\_ \;:=\; lhsOp\_\_\_\_ \otimes b^{n+m} \otimes rhsOp\_\_\_\_$$

We can see that this dynamic rule acts in a flat manner by the following example.

```
In[62]:= foo[c ⊗ a ⊗ a^2 ⊗ b]

Out[62]= foo[c ⊗ a^3 ⊗ b]
```

The pattern reconstruction manipulations given in the *Prototypes* package properly account for the usual pattern matching constructs, such as `Conditions`, `HoldPatterns`, etc. For instance, consider the following conditional associative dynamic assignment.

```
In[63]:= goo ↗ (β_ · α_ /; α ≺lex β) :=a= α · β
```

By examining the information associated with goo, we can see that the condition has been wrapped outside the overall extended pattern on the left hand side.

```
In[64]:= ?? goo

         Global`goo

         goo has the following environment rules :
```

```
goo ↗ (Prototypes`Private`lhsOp___ · β_ ·
            α_ · Prototypes`Private`rhsOp___ /; ! (β ≼tex α)) :=
   Prototypes`Private`lhsOp · α · β · Prototypes`Private`rhsOp
```

It is evident that this dynamic rule acts in a "flat" manner.

```
In[65]:= goo[b · a · j · a]

Out[65]= goo[a · a · b · j]
```

Regrettably, not all pattern matching constructs are extendable. For instance, if the head of the left hand side is the matching construct `Pattern`, then it is not currently possible to transform the pattern to an "extended" pattern.

```
In[66]:= boo ↗ (patt : (β_ · α_) /; α ≺tex β) :=a α · β

DynamicSetDelayed::parseAssociativeFail :
   Pattern cannot be used as the head of the left hand side
         patt : (β_ · α_) /; ! (β ≼tex α) in a Associative dynamic rule
         or assignment. Using default interpretation instead.
```

Technical Note: Actually, it might be possible but it would require some rather masterful contortions, whereas it is much easier to just recreate the rule in a compliant way.

We can see that the dynamic rule actually entered into our system is the same as the one above without any special translations.

```
In[67]:= ?? boo

Global`boo

boo has the following environment rules :

boo ↗ NonCommutativeTimes /: (patt : (β_ · α_) /; ! (β ≼tex α)) := α · β
```

By using the notational methods presented in §2 *The Notation Package* and §3 *Foundations of Notation*, each "associative" dynamic structure is translated into a corresponding new data type having the suffix `Associative`. These associative structures are: `DynamicRuleDelayedAssociative`, `DynamicRuleAssociative`, `DynamicSetDelayedAssociative`, `DynamicSetAssociative`, `DynamicUnsetAssociative`, `TaggedRuleDelayedAssociative`, `TaggedRuleAssociative`, `TaggedSetDelayedAssociative`, `TaggedSetAssociative`, `TagUnsetAssociative`. Here is an example showing the parsing of these special structures.

```
In[68]:= Hold[boo ↗ (patt : (β_ · α_) /; α ≺tex β) :=a α · β] // FullForm

Out[68]//FullForm=
Hold[DynamicSetDelayedAssociative[Condition[
         Pattern[patt, NonCommutativeTimes[Pattern[β, Blank[]],
               Pattern[α, Blank[]]]], Not[OrderedQ[List[β, α]]]],
      NonCommutativeTimes[α, β], NonCommutativeTimes, boo]]
```

These "associative" structures are non-persistent in that they are immediately evaluated and transformed into fuller, dynamic or tagged, structures. For instance,

```
In[69]:= boo ↗ (β_ · α_ /; α ≺tex β) :→a α · β
```

```
Out[69]= boo ↗ (Prototypes`Private`lhsOp___ · β_ · α_ ·
                    Prototypes`Private`rhsOp___ /; ! (β ≼lex α)) :↦
          Prototypes`Private`lhsOp · α · β · Prototypes`Private`rhsOp
```

The implementation of the notations and the pattern translations necessary to cover such rules has previously been treated in §2 *The Notation Package* and §3 *Foundations of Notation*. The interested reader should consult the full code for further details.

It should be mentioned that if one uses a head that *is* `Flat` in an associative rule or an associative assignment, then essentially the associativeness is ignored and the normal flatness takes precedence. Thus the associative structures are in some sense compatible with `Flat` structures.

Before we close this subsection, let us clear all the example functions we have created herein.

```
In[70]:= ClearAll[foo, goo, boo];
```

Henceforth, we almost exclusively use associative structures to state our rules and specifications in preference to a more fully patterned but less readable rule.

5.3.8 Non-Commutative Times III

We have now efficiently changed the underlying implementation of our generic multilinear operators. Let us move on to the combined incorporation of the simplification of powers outlined in §5.3.6 *Non-Commutative Times II* and the developments of §5.3.7 *"Associative" Structures and Associativity*.

First, we need to remove the single rule for the "one identity" nature of non-commutative times.

```
In[71]:= UnAssign [GenericOneIdentity_Rules /. {Generic_Op → NonCommutativeTimes}]
```

In its place, we simply add a slightly updated rule to make sure the head of the expression being reduced is not power.

```
In[72]:= (a_ /; Head@ a_u ≠ Power)_nc := a
```

We also handle the case when all terms in the non-commutative product are taken out.

```
In[73]:= NonCommutativeTimes[] = 1;
```

Let us now proceed to add the rules for the simplification of powers. These can be fairly succinctly and generically stated as follows.

```
In[74]:= GenericPowerSimplification_Rules = {
            Generic_Env ↗ Generic_Op[b_^m_. , b_^n_.] :↦a b^(n+m),
            Generic_Op /: Generic_Op[a_^n_Integer]^m_Integer :↦ Generic_Op[a^(n m)]};
```

As we have done in the past, we add to our rules for NonCommutativeTimes by inheriting from a generic prototype, namely, the one above.

```
In[75]:= Assign [ GenericPowerSimplification_Rules /. {
                    Generic_Op → NonCommutativeTimes,
                    Generic_Env → Simplify}]
```

We also introduce a corresponding set of expansion rules for powers.

```
In[76]:= GenericPowerExpansion_Rules = {
           Generic_Env ↗ Generic_Op[a___, b_^m_Integer?Positive, c___] :>
             (Generic_Op[a, ##, c] & @@ Table[b, {m}])};
```

Similarly, we add these power expansion properties to our rules for NonCommutativeTimes by inheriting from the above generic prototype a generic prototype.

```
In[77]:= Assign [ GenericPowerExpansion_Rules /. {
                    Generic_Op → NonCommutativeTimes,
                    Generic_Env → Expand}]
```

Let us give a few illustrative examples demonstrating the additional behavior our functions now have with powers.

```
In[78]:= Expand [x · (-2 t^2) · (k + p^2)^2]

Out[78]= -2 k^2 x · t · t - 4 k x · t · t · p · p - 2 x · t · t · p · p · p · p

In[79]:= Simplify @ %

Out[79]= -2 (k^2 x · t^2 + 2 k x · t^2 · p^2 + x · t^2 · p^4)
```

We should note that the standard combinations of these operators work.

```
In[80]:= a · (k c · d)

Out[80]= a · (k c · d)

In[81]:= Simplify @ %

Out[81]= k a · c · d
```

There is one unfortunate detail which we must now explain. By examining the rules above, we can see that *Generic*PowerExpansion rules expand powers, yet the *Generic*PowerSimplification rules collect powers. These rules are in conflict. Fortuitously, by the way we have designed the system, if we perform Expand @ Simplify @ *expr* or Simplify @ Expand @ *expr*, we should not encounter conflicts due to the inside argument evaluating completely. This is demonstrated by the following.

```
In[82]:= Simplify @ Expand [x · (-2 t^2) · (k + p^2)^2]

Out[82]= -2 (k^2 x · t^2 + 2 k x · t^2 · p^2 + x · t^2 · p^4)

In[83]:= Expand @ Simplify [x · (-2 t^2) · (k + p^2)^2]
```

```
Out[83]= -2 k^2 x · t · t - 4 k x · t · t · p · p - 2 x · t · t · p · p · p · p
```

However, in some select situations, the *internals* of `Simplify` *call the top level* `Expand`. This clandestine behavior is troubling, since it means we have to check our expansion rules to see if we are in the "simplification environment". Really, this is something of a bug in *Mathematica* and is independent of the *Assign* package. It should be perfectly permissible to have definitions in `Expand` which conflict with those in `Simplify` since these functions are diametrically opposed to each other. As it stands, `Simplify` can suffer a massive slow down since it can call `Expand` many times. Fortunately, there is a way we can tell if we are in the simplification environment. Recall that when a dynamic assignment is active, the calling function gets overridden dynamically — see §4.7.5 *Underpinnings of Dynamic Assignments*. We can view this overriding rule as follows.

```
In[84]:= DownValues @ Expand

Out[84]= {Expand[Assign`Private`expr$___] /;
            Assign`Private`dynamicExpandRulesInactive &&
              Assign`Private`dynamicSimplifyRulesInactive :>
           Assign`Private`evaluateExpandWithDynamicRules [
            Assign`Private`expr$]}
```

As was described in §4.7.5, `Assign`Private`dynamicSimplifyRulesInactive` is always true unless we are inside the simplification environment. The solution we adopt is to change the condition of the override so that the expansion rules will only ever be called if (*i*) the expansion rules are inactive, and (*ii*) the simplification rules are inactive. This is accomplished as follows.

```
In[85]:= Unprotect @ Expand;
         DownValues @ Expand = {Expand[Assign`Private`expr$___] /;
             Assign`Private`dynamicExpandRulesInactive ∧
              Assign`Private`dynamicSimplifyRulesInactive :>
            Assign`Private`evaluateExpandWithDynamicRules [
             Assign`Private`expr$]} ;
         Protect @
          Expand;
```

The following demonstrates that our overall language modifications still work.

```
In[88]:= Simplify @ Expand [ x · (-2 t^2) · (k + p^2)^2 ]

Out[88]= -2 (k^2 x · t^2 + 2 k x · t^2 · p^2 + x · t^2 · p^4)
```

This is somewhat ugly, but it is the price we must pay for *Mathematica* calling the top level command `Expand` from the internals of `Simplify`. This is the only case the author is aware of where conflict arises due to apparent factors outside one's control.

There are various permutations and combinations of these rules which we could use under different environments. That is, we could have power simplification always active, or active only under some grand-expansion routine, etc. Of course, these can all be changed/tailored to

the user's requirements. Indeed, we will use specifically tailored versions as we progress throughout the coming sections.

5.4 Example: The Harmonic Oscillator

5.4.1 The Harmonic Oscillator

Let us start by giving a simple example that is absolutely standard in quantum mechanics, that of the harmonic oscillator. The reason we give such a simple example here is that it illustrates in a familiar setting the notations, language modifications, and generic structures we have developed in this thesis so far. The harmonic oscillator is presented in almost all treatments of quantum mechanics — for example, see [55, 68, 233, 292, etc.]. The reason the harmonic oscillator is so important is that it occurs in many systems throughout quantum mechanics and quantum field theory. For instance, phonons in solid state physics[12, 191], and the radiation field in quantum electrodynamics [69], are both modeled as a collection of harmonic oscillators. The harmonic oscillator is classically governed by the following Hamiltonian.

$$\mathcal{H} = T + V = \frac{p^2}{2m} + \frac{m\omega^2 x^2}{2} \tag{5.4.a}$$

Hamilton's equations for the Hamiltonian in (5.4.a) are $\dot{x} = \partial_p \mathcal{H} = p/m$ and $\dot{p} = -\partial_x \mathcal{H} = -m\omega^2 x$. By eliminating $\dot{p}$, these equations reduce to $m\ddot{x} + m\omega^2 x = 0$. This is, of course, the equation for the standard harmonic oscillator. This system is quantized, as is standardly done in quantum mechanics, by canonical quantization. That is, the classical position and momentum variables are promoted to operators.

$$H = \mathcal{H}(x \to X, p \to P) = \frac{P^2}{2m} + \frac{m\omega^2 X^2}{2} \tag{5.4.b}$$

The operators X and P obey the canonical commutation relation $[X, P] = i\hbar$. The quantum-mechanical oscillator must also obey the Schrödinger equation,

$$i\hbar\,\partial_t\,|\psi\rangle = H\,|\psi\rangle \tag{5.4.c}$$

There are two ways to proceed. (*i*) We could project the energy eigenvalue equation onto the X basis and solve the resulting differential equation by a series expansion, arriving at a criterion that the solutions must satisfy; or (*ii*) we could use creation and annihilation operators. In the next subsection we will briefly illustrate some points about method (*i*), while the subsequent subsections will discuss (*ii*) in more depth.

Before we start our discussions proper, let us dispense with a few preliminaries. In what follows, we will need the tensorial notation developed in §3.4 *Tensorial Notation*. This is obtained by loading the *Tensors* package. However, we do not specifically use any of the tensorial canonicalization routines contained therein until after §6 *An Algorithm for Tensor Simplification*. We do not have to explicitly load the *Tensors* package since it is automatically loaded by the *Prototypes* package.

The `DeclareIndexClass::overridingContexts` warning arises from the fact that some of the dummy indices used in the calculations of the preceding subsections are needed by the *Tensors* package for indices. Consequently, these symbols have been removed from the global context. Also, since we will be dealing with concepts and examples in quantum mechanics, let us also load the notations for bras, kets, operators, and eigenlabels.

```
In[1]:= << Notations`BraKet`
```

5.4.2 Eigenbasis Projection

In this subsection we present a small amount of machinery that will allow projection of eigenbras onto operator equations. We could set up the following discussions in a more general setting, but since our intended purpose is illustration, let us forgo total generality.

We know that the matrix elements of the momentum operator P in the X basis are differential operators. The passage from one basis to the other is governed by a Fourier transform — see any book in quantum mechanics. Let us encode this knowledge. First, we need a non-evaluating generic partial differentiation operator. Here is the notation for such an object.

```
In[2]:= Notation[∂̂ ⟺ operatorPD]
```

We then wrap this operator symbol in a tensor wrapper, with the index representing the variable of differentiation. Here is a simple example of this operator.

```
In[3]:= ∂̂_x
```

$$\text{Out[3]=}\ \hat{\partial}_x$$

```
In[4]:= FullForm @ %
Out[4]//FullForm=
        Tensor[operatorPD, List[Low[x]]]
```

Technical Note: The reason why we use a tensor wrapper is that the partial derivative operator is extremely aggressive in regrouping its structure. We could circumvent this by using a grid box as the primary underlying structure, as was done in §5.2.4 *Generic Differentiation*; but since we will be using indexed partial derivatives shortly, we have avoided this.

Let us call the environment that moves a bra through a non-commutative product the *projection environment*. Since in the X basis $P \to -i\,\hbar\,\partial_x$, or equivalently $\langle x|\cdot P\cdot|x'\rangle = -i\,\hbar\,\delta(x-x')\,\partial_{x'}$, we essentially have the following relation: $\langle x|P = -i\,\hbar\,\partial_x\,\langle x|$. For computational purposes, we

have suppressed the resolution of identity and the attendant integral. We encode this relation as follows.

```
In[5]:= Projection ↗ ⟨x_x̂ | · p̂ :=a -i ħ ∂̂_x · ⟨x_x̂ |
```

Technical Note: Actually, there is a degree of freedom we have omitted from the above. It turns out that if one X basis is related to another, $\tilde{X}$, by a phase, then we can add the derivative of the phase to our differential operator — see Cohen-Tannoudji[68] or Shankar[292].

Also, the projection through the X operator by an X bra occurs as follows.

```
In[6]:= Projection ↗ ⟨x_x̂ | · x̂ :=a x · ⟨x_x̂ |
```

In the physicist's textbook version, all the subscripts of the variables have been suppressed since the physicist intuitively knows what they are — see the comment on lexically scoping in bras and kets in §2.7.4 *Example Calculations from Physics*. Moreover, physicists do not explicitly include a non-commutative times, even though it is implied. Finally, they of course do not use the notation $:\overset{a}{=}$ for assignment, or include environment wrappers in their statements, or include patterns in their rules, as we must do to ensure proper functioning. Other than these provisos, we have created the notation that physicists would use. Certainly it should be recognizable to a physicist.

Technical Note: There are alternate names by which we could have denoted the `Projection` environment. Amongst others, we might have used 'resolution', or 'quantization'. We have chosen the given name since in quantum mechanics we speak of "projecting an equation onto the X basis", which is achieved by multiplying by the basis bra and inserting resolution of identity pair products.

We also want the `Projection` environment to have the same expansion properties over non-commutative time operations as does `Expand`. Thus, we inherit these properties.

```
In[7]:= Assign[Select[Values @ Expand, NonCommutativeTimes ∈?∞ rule &_rule] /.
          Expand → Projection]
```

Let us now observe this basic projection environment working in a simple example.

```
In[8]:= ⟨x_x̂ | · p̂ · x̂ · |ψ⟩
Out[8]= ⟨x_x̂ | · p̂ · x̂ · |ψ⟩

In[9]:= Projection @ %
Out[9]= Projection[-i ħ ∂̂_x · x · ⟨x_x̂ | · |ψ⟩]
```

The X basis eigenbras acting on a ket should just give that wavefunction in the X basis. This point could be elaborated on at length, since there are some rather subtle ideas in play here. Typically, most physics books gloss over this point — see Cohen-Tannoudji[68] for a more detailed discussion. Instead, since we are only providing illustrative examples, let us define the rather specific projection of the energy eigenfunctions onto the X basis, that is $\langle x \mid \mathcal{E} \rangle = \psi_{\mathcal{E}}(x)$.

```
In[10]:= Projection ↗ ⟨x_x̂ | · |ℰ_ĥ⟩ :=a ψ_ℰ[x]
```

Finally, it remains to declare that the `Projection` operator, apart from its environmental values, has no other action.

```
In[11]:= Projection @ any_ := DynamicBar @ any
```

Let us now present a simple example of a projection onto the X basis.

```
In[12]:= ⟨x_X̂ | · P̂ · X̂ · | ε_Ĥ⟩ // Projection
```

$$\text{Out[12]= } -\mathrm{i}\,\hbar\,\hat{\boldsymbol{\partial}}_x \cdot x \cdot \psi_\varepsilon[x]$$

Since the `Projection` environment inherited all of the relevant expansion properties over non-commutative operations, it follows that powers, sums etc., are all handled, correctly and consistently.

```
In[13]:= ⟨x_X̂ | · (P̂^2 + X̂ + P̂ · X̂) · | ε_Ĥ⟩ // Projection
```

$$\text{Out[13]= } x \cdot \psi_\varepsilon[x] - \mathrm{i}\,\hbar\,\hat{\boldsymbol{\partial}}_x \cdot x \cdot \psi_\varepsilon[x] - \hbar^2\,\hat{\boldsymbol{\partial}}_x \cdot \hat{\boldsymbol{\partial}}_x \cdot \psi_\varepsilon[x]$$

It is worthwhile noting that this operation is quite fast.

```
In[14]:= ⟨x_X̂ | · (P̂^2 + X̂ + P̂ · X̂) · | ε_Ĥ⟩ // Projection // Timing
```

$$\text{Out[14]= } \{0.0333333\ \text{Second},\ x \cdot \psi_\varepsilon[x] - \mathrm{i}\,\hbar\,\hat{\boldsymbol{\partial}}_x \cdot x \cdot \psi_\varepsilon[x] - \hbar^2\,\hat{\boldsymbol{\partial}}_x \cdot \hat{\boldsymbol{\partial}}_x \cdot \psi_\varepsilon[x]\}$$

Let us return to the harmonic oscillator. The Hamiltonian is given as follows.

```
In[15]:= Ĥ = (P̂^2)_nc / (2 m) + m ω^2 (X̂^2)_nc / 2
```

$$\text{Out[15]= } \frac{\hat{P}^2{}_{nc}}{2\,m} + \frac{1}{2}\,m\,\omega^2\,\hat{X}^2{}_{nc}$$

For our calculations, we must declare `m`, ω, ε to be constants.

```
In[16]:= DeclareConstant[{m, ω, ε}]
```

The energy eigenvalue equation is $H|\mathcal{E}_H\rangle = \mathcal{E}|\mathcal{E}_H\rangle$. If we project this equation onto the X basis, we obtain the following.

```
In[17]:= ⟨x_X̂ | · Ĥ · | ε_Ĥ⟩ == ⟨x_X̂ | · ε · | ε_Ĥ⟩
```

$$\text{Out[17]= } \left\langle x_{\hat{X}} \right| \cdot \left(\frac{\hat{P}^2{}_{nc}}{2\,m} + \frac{1}{2}\,m\,\omega^2\,\hat{X}^2{}_{nc} \right) \cdot \left| \varepsilon_{\hat{H}} \right\rangle == \left\langle x_{\hat{X}} \right| \cdot \varepsilon \cdot \left| \varepsilon_{\hat{H}} \right\rangle$$

```
In[18]:= % // Projection
```

$$\text{Out[18]= } \frac{1}{2}\,m\,\omega^2\,x \cdot x \cdot \psi_\varepsilon[x] - \frac{\hbar^2\,\hat{\boldsymbol{\partial}}_x \cdot \hat{\boldsymbol{\partial}}_x \cdot \psi_\varepsilon[x]}{2\,m} == \varepsilon\,\psi_\varepsilon[x]$$

We could provide further tools for making our partial derivative operators act on the eigenfunctions, but for now we postpone this kind of operation to §5.4.4 *Creation and Annihilation Operators*. The physics texts will go on to solve this differential equation via a series solution. By doing this series expansion, they arrive at a criterion that $\mathcal{E}$ must satisfy in order for the solutions to converge. The criterion is that $\mathcal{E} = (n + 1/2)\,\hbar\,\omega$ where $n = 0, 1, \ldots$. Thus, we speak of the oscillator being "quantized", that is, it has a discrete quantized set of permissible energies. We have omitted from our discussion this last part — the solving of the above differential equation. This last part is rather standard in physics texts [55, 68, 233, 292], and indeed texts on quantum mechanics based around computer algebra systems [107, 163]. However, setting up the equations in the consistent manner in which we did, and having the tools to properly manipulate such equations, is unique. Both Feagin and Horbatsch jump straight to the differential equation without setting up the problem in a more general context. For instance, we could also introduce similar projection rules for momentum space variables as follows.

```
In[19]:= Projection ↗ ⟨p_p̂ | · P̂ :=ᵃ p · ⟨p_p̂ |
```

```
In[20]:= Projection ↗ ⟨p_p̂ | · X̂ :=ᵃ i ħ ∂̂_p · ⟨p_p̂ |
```

```
In[21]:= Projection ↗ ⟨p_p̂ | · |ℰ_Ĥ⟩ :=ᵃ ψ_ℰ[p]
```

Using these relations, we can project the energy eigenvalue equation onto the momentum basis. We obtain the following differential equation for the energy eigenfunctions in the momentum basis.

```
In[22]:= ⟨p_p̂ | · Ĥ · |ℰ_Ĥ⟩ == ⟨p_p̂ | · ℰ · |ℰ_Ĥ⟩ // Projection
```

$$\text{Out[22]=}\quad \frac{p \cdot p \cdot \psi_{\mathcal{E}}[p]}{2\,m} - \frac{1}{2}\,m\,\omega^2\,\hbar^2\,\hat{\partial}_p \cdot \hat{\partial}_p \cdot \psi_{\mathcal{E}}[p] == \mathcal{E}\,\psi_{\mathcal{E}}[p]$$

This whole process can become tricky when, say, we are projecting a complex Hamiltonian down onto say spherical coordinates. It is worth pointing out that since we have created properly lexically scoped variables, we can substitute any explicit variable into our basis eigenbra. For instance, let us use the bra $\langle y|$ in the X basis.

```
In[23]:= ⟨y_X̂ | · Ĥ · |ℰ_Ĥ⟩ == ⟨y_X̂ | · ℰ · |ℰ_Ĥ⟩ // Projection
```

$$\text{Out[23]=}\quad \frac{1}{2}\,m\,\omega^2\,y \cdot y \cdot \psi_{\mathcal{E}}[y] - \frac{\hbar^2\,\hat{\partial}_y \cdot \hat{\partial}_y \cdot \psi_{\mathcal{E}}[y]}{2\,m} == \mathcal{E}\,\psi_{\mathcal{E}}[y]$$

This subsection has demonstrated the use of our notations and inheritance paradigm as applied to the setting up of a simple problem of the projection of eigenbras in quantum mechanics. Typically, in a quantum mechanics text, the issue we just focused on is usually mentioned only briefly and occupies a small fraction of any illustrative example. However, consistently handling this process is insightful for us since it not only illustrates the inheritance paradigm, language modifications, and generic prototypes, but it also gives a paradigmatic example of how we would handle this process in other settings. This is an important point, since it is highly beneficial to set up our working paradigm to be as consistent and as uniform as possible.

Let us next examine the harmonic oscillator using creation and annihilation operators.

5.4.3 Normal Ordering and Hamiltonian Factorization

Whereas the methods of the previous subsection focused on certain aspects of the treatment of the harmonic oscillator via differential equations, we now present the treatment via creation and annihilation operators. Using creation and annihilation operators to treat the harmonic oscillator is standard in texts on quantum mechanics, but it allows us to present further illustrations and manipulations of non-commutative objects. Our treatment will cover several subsections as we present the various aspects involved in their manipulation and handling.

Since the operators X and P are related by $[X, P] = i\hbar$, we can introduce a simple function for transforming any product of the form $P \cdot X$ to $X \cdot P - i\hbar$. This leads to what is called *"normal ordering"*, and it is a standard concept in physics [173, 186, 206, etc.]. We will use the concept of normal ordering in many places throughout the rest of this thesis.

We would like our normal ordering operation, `NormalOrder`, to have the same expansion rules as `Expand` over non-commutative products. Thus, we inherit these rules first.

```
In[24]:= Assign[Select[Values @ Expand, NonCommutativeTimes ∈₁^∞ rule& &_rule] /.
           Expand → NormalOrder]
```

Now we simply add the normal ordering property to the environment `NormalOrder`.

```
In[25]:= NormalOrder ↗ (P̂ · X̂)_hg :=^a X̂ · P̂ - i ħ
```

Slightly differently than before, upon completion we would like the result of a normal ordering to be simplified.

```
In[26]:= NormalOrder @ any_ := DynamicBar @ Simplify @ any
```

Let us now demonstrate the normal ordering of an expression in X and P.

```
In[27]:= NormalOrder[P̂ · X̂]
Out[27]= -i ħ + X̂ · P̂

In[28]:= NormalOrder[P̂ · X̂^2 · (P̂ + m X̂)]
Out[28]= -3 i m ħ X̂^2_nc - 2 i ħ X̂ · P̂ + X̂^2 · P̂^2 + m X̂^3 · P̂
```

At this stage, it is typical in physics texts to introduce the creation and annihilation operators. These operators are a carefully chosen mix of the position and momentum operators. They have special properties which we soon show. First, we create the notation for such objects.

```
In[29]:= Notation[a⁺ ⟺ CreationOp[a]]

In[30]:= Notation[a⁻ ⟺ AnnihilationOp[a]]
```

It is worth pointing out that this notation differs slightly from that used in most texts — see [173, 186, 206, 292, etc.]. Instead of using a and $a^\dagger$ we have used a⁻ and a⁺. The reason for our slight modification is that using an "unadorned" annihilation operator, and at the same time distinguishing it from the letter 'a' is awkward. To accomplish this distinction, we would have to introduce hidden tag boxes akin to those used in §2.6.6 *Changing Precedences and the Option SyntaxForm*. Thus, we settle on the completely unambiguous use of a⁻ and a⁺. In addition, this maintains consistency with the standard notations for the creation and annihilation operators for angular momentum, that is J_+ and J_-.

Technical Note: The introduction of the above notation for creation and annihilation operators raises another point. Recall that we used i^+ and i^- to refer to high and low indices, in our notation for tensors — see §3.4.3 *Prototypical Tensor Expression Structure*. This is in conflict with our notations for creation and annihilation operators. For instance, is a^+ a high index 'a' or is it a creation operator? Here is a case were it would be ideal to have "generalized" context sensitive notations. For now we resolve this issue by blithely using a^+ and a^- as creation and annihilation operators.

Let us introduce the expression for the creation and annihilation operators in terms of the standard basis operators.

```
In[31]:= BasisForm ↗ a⁻ := Sqrt[m ω / (2 ħ)] X̂ + i Sqrt[1 / (2 m ω ħ)] P̂
         BasisForm ↗ a⁺ := Sqrt[m ω / (2 ħ)] X̂ - i Sqrt[1 / (2 m ω ħ)] P̂
```

We wold also like `BasisForm` to have the normal ordering properties above, so we inherit these.

```
In[33]:= Assign[Values @ NormalOrder /. NormalOrder → BasisForm]
```

Finally, for now, we do not simplify the result of expressing the creation and annihilation operators in terms of the position and momentum operators.

```
In[34]:= BasisForm @ any_ := DynamicBar @ any;
```

The reason for introducing the creation and annihilation operators in the way that we did is that they have special properties in this form. The first such property we show involves the commutation of these operators.

```
In[35]:= [a⁻, a⁺]_ // Expand // BasisForm
```

$$\text{Out[35]=}\quad -i\sqrt{\frac{1}{m\omega\hbar}}\sqrt{\frac{m\omega}{\hbar}}\,\hat{X}\cdot\hat{P} + i\sqrt{\frac{1}{m\omega\hbar}}\sqrt{\frac{m\omega}{\hbar}}\,(-i\hbar + \hat{X}\cdot\hat{P})$$

```
In[36]:= Simplify[%, m > 0 ∧ ω > 0]
Out[36]= 1
```

Thus we explicitly see that see a⁻ and a⁺ have been constructed so that their commutator is the identity, that is, in traditional notation, $[a, a^\dagger] = 1$. Moreover, The operators a⁻ and a⁺ have the important property that they are simply related to the Hamiltonian of our system.

$$\text{In[37]:=}\quad \hat{H} == \left(a^+ \cdot a^- + \frac{1}{2}\right)\hbar\,\omega$$

```
In[37]:= (1/(2 m)) P^2_nc + (1/2) m ω^2 X^2_nc == ω ħ (1/2 + a^+ · a^-)
```

```
In[38]:= BasisForm @ %
```

$$\text{Out[38]= } \frac{\hat{P}\cdot\hat{P}}{2\,m}+\frac{1}{2}\,m\,\omega^2\,\hat{X}\cdot\hat{X}==\omega\,\hbar\left(\frac{1}{2}+\frac{\hat{P}\cdot\hat{P}}{2\,m\,\omega\,\hbar}+\frac{1}{2}\,i\,\sqrt{\frac{1}{m\,\omega\,\hbar}}\,\sqrt{\frac{m\,\omega}{\hbar}}\,\hat{X}\cdot\hat{P}-\frac{1}{2}\,i\,\sqrt{\frac{1}{m\,\omega\,\hbar}}\,\sqrt{\frac{m\,\omega}{\hbar}}\,(-i\,\hbar+\hat{X}\cdot\hat{P})+\frac{m\,\omega\,\hat{X}\cdot\hat{X}}{2\,\hbar}\right)$$

```
In[39]:= Simplify[%, m > 0 ∧ ω > 0]
```

```
Out[39]= True
```

Thus we see that the Hamiltonian is expressible in terms of the creation and annihilation operators. That is, ignoring constants, $H = X^2 + P^2 = (X - iP)(X + iP) + 1/2$, or we have "factorized" the Hamiltonian. The next subsection looks at formulating the workings of our quantum mechanical oscillator in terms of creation and annihilation operators.

Technical Note: In §5.3.8 *Non-Commutative Times III*, we stated that there were certain select cases where the internals of `Simplify` called `Expand`. The last example above is one such case where this occurs. Recall that to circumvent this, we had to change our dynamic rule condition to ensure that the expansion rules were never used once the simplification rules were being used.

5.4.4 Creation and Annihilation Operators

As with the other examples so far, the designer has to decide at which specific point he/she would like to demarcate the functionality. In our present case, this requires us to decide whether `BasisForm` also includes simplification along with its normal ordering and its re-expression of creation and annihilation operators? If one so desires, this can easily be added. For instance, in the above examples it might have been convenient to have `BasisForm` perform simplification at the last stage of its action. (We previously avoided doing this since for the purposes of illustration, it was better to see the explicit steps taking place.)

```
In[40]:= BasisForm @ any_ := DynamicBar @ Simplify[any, m > 0 ∧ ω > 0];
```

So, even though it is slightly repetitious, it is nice to note that with the addition of our simplification to `BasisForm`, the Hamiltonian relation drops out in a single line.

```
In[41]:= Ĥ == (a^+ · a^- + 1/2) ħ ω // BasisForm
```

```
Out[41]= True
```

As a preliminary to showing how a^- and a^+ act on eigenkets, let us examine the commutation relations of these operators with the Hamiltonian.

```
In[42]:= [a^-, Ĥ]_ == ħ ω a^- // Expand // BasisForm
```

```
Out[42]= True
```

```
In[43]:= [a⁺, Ĥ]_ == -ħ ω a⁺ // Expand // BasisForm
Out[43]= True
```

Henceforth, let us switch to using the creation and annihilation formulation for all of our operations. To enact this change to creation and annihilation operators, we need to specify how these operators normal order.

```
In[44]:= NormalOrder ↗ a⁻ · a⁺ :=ᵃ a⁺ · a⁻ + 1
```

Recall from above that $[a, a^\dagger] = 1$.

```
In[45]:= [a⁻, a⁺]_ // Expand // BasisForm
Out[45]= 1
```

Our normal ordering rule is consistent with this commutator.

```
In[46]:= [a⁻, a⁺]_ // Expand // NormalOrder
Out[46]= 1
```

It is convenient to define a new environment where the Hamiltonian is normal ordered with respect to the creation and annihilation operators. In addition this environment will also allow the Hamiltonian to act on eigenkets. Let us reuse the environment `Act` for this purpose. We first clear `Act`, and then inherit the rules of `NormalOrder`.

```
In[47]:= Ĥ = .;
         ClearAll[Act];
         Assign[Values @ NormalOrder /. NormalOrder → Act]
```

Next, let us encode the above commutation relations between the transformed Hamiltonian and the creation and annihilation operators.

```
In[50]:= Act ↗ (Ĥ · a⁻)_hp :=ᵃ a⁻ · Ĥ - ħ ω a⁻
         Act ↗ (Ĥ · a⁺)_hp :=ᵃ a⁺ · Ĥ + ħ ω a⁺
```

As the final preliminary, let us add the action of the Hamiltonian on the eigenkets, much as the environment `Projection` acted in §5.4.2 *Eigenbasis Projection*.

```
In[52]:= Act ↗ (Ĥ · |ε_ĥ⟩)_hp :=ᵃ (ε + 1 / 2) ħ ω |ε_ĥ⟩
```

The ground state energy is given by $\hbar\omega/2$, as it should be.

```
In[53]:= Ĥ · |0_ĥ⟩ // Act
Out[53]= 1/2 ω ħ |0_ĥ⟩
```

The creation operator acting on an eigenket yields a new eigenket according to the following.

```
In[54]:= Ĥ · a⁻ · |ε_ĥ⟩ // Act
```

```
Out[54]= 1/2 (-1 + 2 ε) ω ħ a⁻ · |ε_ĥ⟩
```

So $a|\mathcal{E}_H\rangle$ is an eigenket of H. Yet $a|\mathcal{E}_H\rangle$ shares the same eigenvalue as $|\mathcal{E}-1_H\rangle$ as we see by the following.

```
In[55]:= Ĥ · |(ε - 1)_ĥ⟩ // Act

Out[55]= (-1/2 + ε) ω ħ |(-1 + ε)_ĥ⟩
```

So we know that $a|\mathcal{E}_H\rangle = C_\mathcal{E}|\mathcal{E}-1_H\rangle$, for some $C_\mathcal{E}$. Similarly, we can show that $a^\dagger|\mathcal{E}_H\rangle = C_{\mathcal{E}+1}|\mathcal{E}+1_H\rangle$. To find the values $C_\mathcal{E}$ and $C_{\mathcal{E}+1}$ is not difficult, and occupies more of the same sort of manipulations that we have just given. Since we pursued such a simple example in the first place only for illustration purposes, let us only quote the results of the raising and lowering operators acting on energy eigenkets. That is, up to a phase, which we choose to be the identity,

$$\begin{aligned} a|n_H\rangle &= \sqrt{n}\,|n-1_H\rangle \\ a^\dagger|n_H\rangle &= \sqrt{n+1}\,|n+1_H\rangle \end{aligned} \tag{5.4.d}$$

We can immediately encode these relations, (5.4.d), into our `Act` environment.

```
In[56]:= Act ↗ a⁻ · |n_ĥ⟩ :=ᵃ √n |(n - 1)_ĥ⟩
         Act ↗ a⁺ · |n_ĥ⟩ :=ᵃ √(n + 1) |(n + 1)_ĥ⟩
```

It remains to find the eigenfunction for the ground state and then, using the ladder operators, we can create the hierarchy of eigenfunctions. The ground state eigenfunction can be found by projecting the ladder equation $a|0_H\rangle = 0$ onto the X basis. We use the `Projection` operator defined in §5.4.2 *Eigenbasis Projection*.

```
In[58]:= ⟨x_x̂| · a⁻ · |0_ĥ⟩ == 0 // BasisForm // Projection

Out[58]= (√(m ω/ħ) x · ψ0[x] + √(1/(m ω ħ)) ħ ∂̂_x · ψ0[x]) / √2 == 0
```

The above differential equation must be solved to find the ground state eigenfunction. This can be accomplished with a straightforward application of `DSolve`, and an integration to find the normalization constant.

```
In[59]:= solution = First @
            First @ DSolve[√(m ω/ħ) x ψ0[x] + √(1/(m ω ħ)) ħ D[ψ0[x], x] == 0, ψ0[x], x];
         solution = Simplify[solution, m > 0 ∧ ω > 0 ∧ ħ > 0]

Out[60]= ψ0[x] → e^(-m x² ω/(2ħ)) C[1]
```

To find the normalization constant we simply solve $\int_{-\infty}^{\infty}\overline{\psi_0(x)}\,\psi_0(x)\,dx = 1$ for `C[1]`. For reference, the normalized solution is

$$\psi_0[x] = e^{-\frac{m x^2 \omega}{2\hbar}} \left(\frac{m\omega}{\pi\hbar}\right)^{1/4} \tag{5.4.e}$$

The other eigenfunctions can then be found by laddering. For instance, if we act on the ground state with a creation operator in both possible ways, and then project the resulting equation onto the X basis, we arrive at the following equation.

```
In[61]:= BasisForm[⟨x_X̂ | · a⁺ · |0_Ĥ⟩] == Act[⟨x_X̂ | · a⁺ · |0_Ĥ⟩] // Projection
```

$$\text{Out[61]=} \quad \frac{\sqrt{\frac{m\omega}{\hbar}}\, x \cdot \psi_0[x] - \sqrt{\frac{1}{m\omega\hbar}}\, \hbar\, \hat{\boldsymbol{\partial}}_x \cdot \psi_0[x]}{\sqrt{2}} == \psi_1[x]$$

These eigenfunctions correspond to Hermite polynomials. Further details are given in almost any text on quantum mechanics.

5.4.5 Computations with Creation and Annihilation Operators

In this subsection we look at performing computations and evaluating matrix elements using the creation and annihilation relations developed in the previous subsection. There are a few preliminary issues we must dispense with. First, we should note that the energy eigenkets are normalized.

```
In[62]:= Act ↗ ⟨m_Ĥ | · |n_Ĥ⟩ :=ᵃ δ_m n
```

Second, we need to be able to express the position and momentum operators in terms of the creation and annihilation operators. That is, we need to be able to express the relations inverse to those embodied in the `BasisForm` relations. Let us add these inverse relations to the `Act` environment.

```
In[63]:= Act ↗ X̂ := Sqrt[ħ/(2 m ω)] · (a⁺ + a⁻)

         Act ↗ P̂ := i Sqrt[m ω ħ/2] · (a⁺ - a⁻)
```

Under the `Act` environment X and P are now expressed in terms of creation and annihilation operators.

```
In[65]:= X̂ · (P̂ + X̂²) // Act
```

$$\text{Out[65]=} \quad \frac{1}{2\sqrt{2}\,\hbar}\left(\left(\frac{\hbar}{m\omega}\right)^{3/2}\left(i\sqrt{2}\, m\omega\sqrt{m\omega\hbar} + 3\hbar a^- + 3\hbar a^+ - i\sqrt{2}\, m\omega\sqrt{m\omega\hbar}\,(a^-)^2_{nc} + \hbar\,(a^-)^3_{nc} + i\sqrt{2}\, m\omega\sqrt{m\omega\hbar}\,(a^+)^2_{nc} + \hbar\,(a^+)^3_{nc} + 3\hbar a^+ \cdot (a^-)^2 + 3\hbar\,(a^+)^2 \cdot a^-\right)\right)$$

Since X and P are now expressed in terms of creation and annihilation operators, it is trivial to perform, say, the computation $\langle 3_H | X | 2_H \rangle$.

```
In[66]:= ⟨3_ĥ | · X̂^2 · | 2_ĥ⟩ // Act
```

$$\text{Out[66]=} \quad \frac{\hbar \left(\sqrt{2}\, \delta_{30} + 5\, \delta_{32} + 2\sqrt{3}\, \delta_{34}\right)}{2\, m\, \omega}$$

We need to declare that the Kronecker delta is a constant, and in addition that $\delta_{ij} = 1$ if $i = j$ and 0 otherwise.

```
In[67]:= DeclareConstant[δ__]

         δ_i_j_ := If[(i - j) ≡ 0, 1, 0] /; IntegerQ[i - j]
```

Technical Note: The above handles simple symbolic arguments like $\delta_{n\,n-1}$, but is not fully general in its semanticness. For instance, indices such as $\sqrt{2} + \sqrt{3} - \sqrt{5 + 2\sqrt{6}}$ will not be recognized as 0, even though they are equivalent to 0 through, say, `RootReduce`.

We have now developed the machinery whereby we can evaluate matrix elements. For instance,

```
In[69]:= ⟨3_ĥ | · X̂^3 · | 2_ĥ⟩ // Act
```

$$\text{Out[69]=} \quad \frac{9}{2}\sqrt{\frac{3}{2}}\left(\frac{\hbar}{m\,\omega}\right)^{3/2}$$

We can easily create tables of matrix elements.

```
In[70]:= Table[⟨i_ĥ | · X̂ · | j_ĥ⟩, {i, 0, 3}, {j, 0, 3}] // TableForm
```

Out[70]//TableForm=

$\langle 0_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 0_{\hat{h}} \rangle$	$\langle 0_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 1_{\hat{h}} \rangle$	$\langle 0_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 2_{\hat{h}} \rangle$	$\langle 0_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 3_{\hat{h}} \rangle$
$\langle 1_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 0_{\hat{h}} \rangle$	$\langle 1_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 1_{\hat{h}} \rangle$	$\langle 1_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 2_{\hat{h}} \rangle$	$\langle 1_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 3_{\hat{h}} \rangle$
$\langle 2_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 0_{\hat{h}} \rangle$	$\langle 2_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 1_{\hat{h}} \rangle$	$\langle 2_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 2_{\hat{h}} \rangle$	$\langle 2_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 3_{\hat{h}} \rangle$
$\langle 3_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 0_{\hat{h}} \rangle$	$\langle 3_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 1_{\hat{h}} \rangle$	$\langle 3_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 2_{\hat{h}} \rangle$	$\langle 3_{\hat{h}} \lvert \cdot \hat{X} \cdot \rvert 3_{\hat{h}} \rangle$

These matrix elements are inert, as they should be, until we put them in an environment where they should become active.

```
In[71]:= Act @ % // TableForm
```

Out[71]//TableForm=

0	$\frac{\sqrt{\frac{\hbar}{m\,\omega}}}{\sqrt{2}}$	0	0
$\frac{\sqrt{\frac{\hbar}{m\,\omega}}}{\sqrt{2}}$	0	$\sqrt{\frac{\hbar}{m\,\omega}}$	0
0	$\sqrt{\frac{\hbar}{m\,\omega}}$	0	$\sqrt{\frac{3}{2}}\sqrt{\frac{\hbar}{m\,\omega}}$
0	0	$\sqrt{\frac{3}{2}}\sqrt{\frac{\hbar}{m\,\omega}}$	0

It is reassuring to see that the Hamiltonian is diagonal in its own basis.

```
In[72]:= Table[⟨i_n̂ | · Ĥ · | j_n̂⟩, {i, 0, 3}, {j, 0, 3}] // Act // TableForm
```

Out[72]//TableForm=

$$\begin{array}{cccc} \frac{\omega\hbar}{2} & 0 & 0 & 0 \\ 0 & \frac{3\omega\hbar}{2} & 0 & 0 \\ 0 & 0 & \frac{5\omega\hbar}{2} & 0 \\ 0 & 0 & 0 & \frac{7\omega\hbar}{2} \end{array}$$

Here are the matrix elements of P^3, where we have removed the factor of $(m\,\omega\,\hbar)^{3/2}$.

```
In[73]:= Table[⟨i_n̂ | · P̂^3 · | j_n̂⟩ / (m ω ħ)^(3/2), {i, 0, 4}, {j, 0, 4}] // Act // TableForm
```

Out[73]//TableForm=

$$\begin{array}{ccccc} 0 & -\frac{3\,i}{2\sqrt{2}} & 0 & \frac{i\sqrt{3}}{2} & 0 \\ \frac{3\,i}{2\sqrt{2}} & 0 & -3\,i & 0 & i\sqrt{3} \\ 0 & 3\,i & 0 & -\frac{9}{2}\,i\sqrt{\frac{3}{2}} & 0 \\ -\frac{i\sqrt{3}}{2} & 0 & \frac{9}{2}\,i\sqrt{\frac{3}{2}} & 0 & -6\,i\sqrt{2} \\ 0 & -i\sqrt{3} & 0 & 6\,i\sqrt{2} & 0 \end{array}$$

Before we close this subsection, let us comment on some of the expectation values for our stationary states. The expectation value of both the momentum and the position is zero.

```
In[74]:= DeclareConstant[n]
```

```
In[75]:= ⟨n_n̂ | · P̂ · | n_n̂⟩ // Act
```

```
Out[75]= 0
```

```
In[76]:= ⟨n_n̂ | · X̂ · | n_n̂⟩ // Act
```

```
Out[76]= 0
```

Yet the expectation values of the squared operators are non-zero.

```
In[77]:= ⟨n_n̂ | · P̂^2 · | n_n̂⟩ // Act
```

$$\text{Out[77]=}\ \frac{1}{2}\,m\,(1+2\,n)\,\omega\,\hbar$$

```
In[78]:= ⟨n_n̂ | · X̂^2 · | n_n̂⟩ // Act
```

$$\text{Out[78]=}\ \frac{\hbar+2\,n\,\hbar}{2\,m\,\omega}$$

In a similar manner, we can calculate the uncertainties of the position and momentum operators.

```
In[79]:= ΔX = √(⟨n_n̂ | · X̂^2 · | n_n̂⟩ - (⟨n_n̂ | · X̂ · | n_n̂⟩)^2) // Act
```

Out[79]= $\frac{\sqrt{\frac{\hbar+2\,n\,\hbar}{m\,\omega}}}{\sqrt{2}}$

In[80]:= $\Delta P = \sqrt{\langle n_{\hat{H}} | \cdot \hat{P}^2 \cdot | n_{\hat{H}} \rangle - (\langle n_{\hat{H}} | \cdot \hat{P} \cdot | n_{\hat{H}} \rangle)^2}$ // Act

Out[80]= $\frac{\sqrt{m\ (1 + 2\,n)\ \omega\ \hbar}}{\sqrt{2}}$

These uncertainties satisfy the Heisenberg uncertainty principle, $\Delta X\,\Delta P \geq \hbar/2$, as they must.

In[81]:= FullSimplify[$\Delta X\,\Delta P$, $m > 0 \wedge \hbar > 0 \wedge \omega > 0 \wedge n > 0$]

Out[81]= $\left(\frac{1}{2} + n\right)\hbar$

Indeed, let us calculate the expected kinetic and potential energies in the quantum harmonic oscillator Hamiltonian, (5.4.b). We do this by evaluating the following.

In[82]:= $\langle n_{\hat{H}} | \cdot \frac{\hat{P}^2}{2\,m} \cdot | n_{\hat{H}} \rangle$ // Act

Out[82]= $\frac{1}{4}\ (1 + 2\,n)\ \omega\ \hbar$

In[83]:= $\langle n_{\hat{H}} | \cdot \frac{m\,\omega^2\,\hat{X}^2}{2} \cdot | n_{\hat{H}} \rangle$ // Act

Out[83]= $\frac{1}{4}\ (1 + 2\,n)\ \omega\ \hbar$

Each term accounts for just half of the total energy of the quantum harmonic oscillator.

In[84]:= $\langle n_{\hat{H}} | \cdot \hat{H} \cdot | n_{\hat{H}} \rangle$ // Act

Out[84]= $\left(\frac{1}{2} + n\right)\omega\ \hbar$

Thus on average, half of the energy is in the kinetic part of the Hamiltonian, and half is in the potential part of the Hamiltonian. This agrees with the virial theorem — see Shankar[292] or Cohen-Tannoudji[68].

Before moving on to simple perturbation theory, let us next briefly tackle a side issue arising from the complexity of commutation of objects with high powers.

5.4.6 Aside: Commutation of Operators Raised to Powers

In the preceding subsections we have made use of the normal ordering properties of specific operators. Due to our general inheritance paradigm, the permissible expressions could contain sums, powers, etc. When these powers are of low degree, the general rules we have developed work perfectly adequately. For example,

```
In[85]:= NormalOrder[P̂^3 · X̂^2]
```

$$\text{Out[85]=}\quad -6\,i\,\hbar\,\hat{X}\cdot\hat{P}^2+\hat{X}^2\cdot\hat{P}^3-6\,\hbar^2\,\hat{P}$$

However, our simple formulation gets progressively worse for larger powers.

```
In[86]:= NormalOrder[P̂^10 · X̂^4] // Timing
```

$$\text{Out[86]=}\quad \{53.0667\ \text{Second},\ 5040\,\hbar^4\,\hat{P}^6_{nc}+2880\,i\,\hbar^3\,\hat{X}\cdot\hat{P}^7-540\,\hbar^2\,\hat{X}^2\cdot\hat{P}^8-40\,i\,\hbar\,\hat{X}^3\cdot\hat{P}^9+\hat{X}^4\cdot\hat{P}^{10}\}$$

This has taken an inordinately long time for such a simple computation, where the answer is also simple. By some clever experimentation and playing around with expressions, it is not terribly difficult to come up with a relation for the general case.

$$P^m\cdot X^n=\sum_{i=0}^{\text{Min}[m,n]}(-[X,P])^i\,\text{Binomial}[\text{Min}[m,n],i]\,X^{n-i}\cdot P^{m-i}\left(\prod_{j=1}^{i}(\text{Max}[m,n]-j+1)\right)\qquad(5.4.f)$$

(Actually, this relation must exist in some book, but it would probably take longer to find it than figure it out *ab initio*.) We generically encode the commutator relation (5.4.f), with $[\hat{q},\hat{p}]$ = *CommutatorValue*, as follows.

```
In[87]:= GenericPowerCommutation_Rules = {
  Generic_Env ↗ (p̂^m_. · q̂^n_. /; IntegerQ[m] ∧ IntegerQ[n])_nc :a:→
    Sum[(-CommutatorValue)^i Binomial[Min[m, n], i]
      q̂^(n-i) · p̂^(m-i) (Product[Max[m, n] - j + 1, {j, 1, i}]), {i, 0, Min[m, n]}]};
```

Technical Note: Actually, we can make the above rule slightly more general, so that one of the particular powers can be symbolic. Indeed, if proper handling of abstract sums were truly integrated, we could allow both powers to be symbolic.

We need to incorporate these new found efficiencies into our system. We could either (*i*) add our rule before the expansion of powers of non-commutative operators, or (*ii*) we could completely remove the rule for expanding out products while normal ordering. To implement (*i*), we would unassign the power expansion, then assign our new behavior, then re-assign power expansion. This can be done as follows.

```
In[88]:= NonCommutativeTimesPowerExpansion_Rules = GenericPowerExpansion_Rules /. {
           Generic_Op → NonCommutativeTimes,
           Generic_Env → NormalOrder};

In[89]:= UnAssign [ NonCommutativeTimesPowerExpansion_Rules ]
         Assign[ GenericPowerCommutation_Rules /. {
           CommutatorValue → i ħ,
           Generic_Env → NormalOrder,
           p̂ → P̂, q̂ → X̂ }]
         Assign [ NonCommutativeTimesPowerExpansion_Rules ]
```

Our new behavior is now used preferentially.

```
In[92]:= NormalOrder[P̂^10 · X̂^4] // Timing

Out[92]= {0.183333 Second,
    5040 ħ^4 P̂^6 + 2880 i ħ^3 X̂ · P̂^7 - 540 ħ^2 X̂^2 · P̂^8 - 40 i ħ X̂^3 · P̂^9 + X̂^4 · P̂^10}
```

This result agrees with our earlier calculation. Yet, our old commutators still function. For instance, here is a commutator of creation and annihilation operators.

```
In[93]:= NormalOrder[(a⁻)^4 · (a⁺)^3] // Timing

Out[93]= {0.4 Second, 24 a⁻ + 36 a⁺ · (a⁻)^2 + 12 (a⁺)^2 · (a⁻)^3 + (a⁺)^3 · (a⁻)^4}
```

Alternatively, we could remove the general power expansion rule completely and instead make sure we encode all our old commutation rules in terms of the new power commutation rulees

```
In[94]:= UnAssign [ NonCommutativeTimesPowerExpansion_Rules ]
```

In this case the only other rule necessary is that of creation and annihilation commutation.

```
In[95]:= Assign[GenericPowerCommutation_Rules /. {
           CommutatorValue → -1,
           Generic_Env → NormalOrder,
           p̂ → a⁻, q̂ → a⁺ }]
```

Now the normal ordering of creation and annihilation operators yields the same results as before, except that calculation times have decreased dramatically.

```
In[96]:= NormalOrder[(a⁻)^4 · (a⁺)^3] // Timing

Out[96]= {0.0666667 Second, 24 a⁻ + 36 a⁺ · (a⁻)^2 + 12 (a⁺)^2 · (a⁻)^3 + (a⁺)^3 · (a⁻)^4}
```

Overall, it is convenient to keep the power expansion rule, so we reintroduce it.

```
In[97]:= Assign [ NonCommutativeTimesPowerExpansion_Rules ]
```

Actually in more generality, if $c = [A, B]$ and $[c, A] = [c, B] = 0$, that is both A and B commute with their commutator, then the following relations can be shown to hold via a power series expansion.

$$\begin{aligned} [A, F(B)] &= c\,F'(B) \\ [B, G(A)] &= -c\,G'(A) \end{aligned} \tag{5.4.g}$$

In the above, $F'(z)$ of course denotes the derivative with respect to z.

Technical Note: At least (5.4.g) holds for any "nice" functions F and G. For the basis for a more mathematically rigorous account, see books in Operator Theory — for example, Dunford and Schwartz[99].

There are many other similar optimizations we could incorporate. For example, if we want to perform the computation $(A+B)^n$ where $[A, B]$ commutes with both A and B, we can find special formulas for this. An instance of such an expansion would be say X^{30} in terms of creation and annihilation operators. To find the expansion, one notes that $e^A\,e^B = e^{A+B+[A,B]/2}$ — see any at least intermediate level book in quantum mechanics, such as Shankar[292]. From this we can substitute $A \to \lambda A$ and $B \to \lambda B$, to arrive at $e^{\lambda A}\,e^{\lambda B} = e^{\lambda A+\lambda B+\lambda^2[A,B]/2}$. Since the commutator factor commutes with the other factors, we can transform the equation to the following.

$$e^{\lambda A}\,e^{\lambda B}\,e^{-\lambda^2[A,B]} = e^{\lambda A+\lambda B} \tag{5.4.h}$$

Using this equation we can expand and compare powers of λ. By considering the factor λ^n in (5.4.h) we can determine that the normally ordered expansion for $(A+B)^n$ is the following.

$$(A+B)^n = n! \sum_{\substack{i,j,k>0 \\ i+j+2k=n}} \frac{A^i}{i!}\,\frac{B^j}{j!}\,\frac{(-[A,B])^k}{k!} \tag{5.4.i}$$

This optimized relation can be added in a similar manner to the relation for the commutation of powers above.

Before we close this subsection, let us make a few final comments. The optimization to normal ordering presented in this subsection highlights an area in which we could further improve the workings of our inheritance paradigm if we had access to the internals of *Mathematica*. We have already inherited several functions that are based on `NormalOrder`. In some ways, which we will not quantify, it would be nice to have the environments based on `NormalOrder`, also receive the updated routines. Such issues are studied extensively in computer science as applied to object-oriented programming. For us, we just note the possibility and likely desirability.

The other point that becomes evident in this subsection is that it would have been nice to insert our more optimized rule before the others. We could have elegantly done this if we had rewrite rule precedences, as briefly hinted at in §4.8.1 *Potential Further Developments*.

Let us progress on with our applications. The next subsection continues topics needed in working towards some simple perturbation theory.

5.4.7 Aside: Hermitian Conjugates

Let us now introduce conjugation operations. This will illustrate inheriting only part of the structures of a multilinear operator as well as defining a new class of "constants". We could override the default `Conjugate` operation, but for illustration we define our own conjugation function. Let us call it `HermitianConjugate`. We use a standard notation for Hermitian conjugation.

```
In[98]:= Notation[expr_† ⟺ HermitianConjugate[expr_]]
```

Hermitian conjugation is close to but not quite a multi-linear operator. It inherits both linear distributivity over sums, and is zero for the zero argument. Moreover, we want this distributivity to always be "active", so we strip out the environment. (In fact, we do this for all of our Hermitian rules.)

```
In[99]:= Assign[StripTheEnvironment[MLZero_Rules +_{} MLDistribution_Rules] /. {
              GenericML_Op → HermitianConjugate,
              GenericPlus → Plus}]
```

Hermitian conjugates also have the non-standard property of reversing the order of their arguments when being distributed over, that is $(A\,B)^\dagger = B^\dagger\,A^\dagger$. The same behavior is also true of other structural operations in physics and mathematics, for instance, conjugations, inverses, and transposes amongst others. In category theory there exists the notation of a contravariant functor[216] which in some sense carries out a reversing of the standard order. Based on the lack of a common name for this reversing property, we will denote the general concept by the term *"contra-ordering"*. Here is a generic rule set for such a property.

```
In[100]:= ContraOrdering_Rules =
            {Generic_Env ↗ GenericContraOrdering_Op @ expr_GenericTimes :↦
              GenericContraOrdering_Op /@ Reverse @ expr};
```

We can use this prototypical rule for the contra-ordering of our non-commutative products under Hermitian conjugation.

```
In[101]:= Assign[StripTheEnvironment @ ContraOrdering_Rules /. {
              GenericContraOrdering_Op → HermitianConjugate,
              GenericTimes → NonCommutativeTimes}]
```

We can also use it for the contra-ordering over `Times`, even though `Times` will commutatively reorder its arguments once again.

```
In[102]:= Assign[StripTheEnvironment @ ContraOrdering_Rules /. {
              GenericContraOrdering_Op → HermitianConjugate,
              GenericTimes → Times}]
```

Also, subject to qualifications, Hermitian conjugation distributes over some standard operations. For instance, `Power` is one such operation.

```
In[103]:= (arg_^n_)^† := (arg^†)^n /; PositiveQ[arg] ∨ IntegerQ[n]
```

We could add such distributivity for many other operations such as `Sin`, `Cos`, `HermiteH`$_n$ etc., for which Hermitian conjugation always distributes over. However, there are other functions besides `Power`, over which we cannot always distribute, for instance, `Sqrt` or `Log`. For specific counter examples, $\sqrt{(-1)^\dagger} = \sqrt{-1} = i$, but $\left(\sqrt{-1}\right)^\dagger = i^\dagger = -i$; and also $\text{Log}\left[(-1)^\dagger\right] = \text{Log}[-1] = i\pi$, whereas $\text{Log}[-1]^\dagger = (i\pi)^\dagger = -i\pi$. Thus we do not include any direct rules for simplifying Hermitian conjugates over "mathematical" functions. One could easily introduce an environment where such simplifications take place (such as simplify, etc).

Finally, if we Hermitian conjugate a pure numeric expression, it is the same as standard conjugation.

```
In[104]:= HermitianConjugate @ num_?NumericQ := Conjugate @ num
```

Let us now illustrate this simple Hermitian conjugation operation.

```
In[105]:= (A + 3 B^2 · T · A)^†

Out[105]= A^† + 3 A^† · T^† · (B^†)^2
```

This example yields the exact results that we desire. However, consider the following example.

```
In[106]:= (e^(i Ĥ t))^†

Out[106]= e^(-i t^† Ĥ^†)
```

We know that the Hamiltonian is Hermitian, and also that the time t is real. How can we specify this? We would like a uniform system of declaring that something is Hermitian, and also testing whether something is Hermitian. We have already developed the exact structures to accomplish this in §5.2.3 *Generic Constants*. Thus we need to inherit the rules for constants, only this time we are using "Hermitian-ness", as opposed to "constant-ness".

```
In[107]:= Assign[GenericConstantQ_Rules /. {
                 GenericConstQ → HermitianQ,
                 GenericTimes → Times,
                 GenericPlus → Plus,
                 DeclareGenericConstant → DeclareHermitian,
                 DeclareGenericUnconstant → DeclareUnHermitian}]
```

There are two rules which we must override in this inherited set. We do this as follows.

```
In[108]:= HermitianQ[arg_?NumericQ] := arg ≡ Conjugate @ arg
          HermitianQ[arg_^num_] := (HermitianQ[arg^num] =
                    HermitianQ[arg_u] ∧ HermitianQ[num_u] ∧
                        (Positive[arg_u] ∨ IntegerQ[num_u]))
```

We can now declare that operators and variables are Hermitian.

```
In[110]:= DeclareHermitian[{Ĥ, t, m, ω, ħ}]
```

```
In[111]:= HermitianQ[e^(Ĥ t)]

Out[111]= True

In[112]:= HermitianQ[e^(i Ĥ t)]

Out[112]= False
```

Technical Note: It seems irregular, even to the author, to declare that a real variable is Hermitian. Indeed, conventionally physicists or mathematicians do not talk of a real variable being Hermitian. Hermitian-ness only applies to operators or matrices. Instead, they will talk of complex conjugation. Yet, for the sake of consistency, we need an overarching operation. We could have chosen the term "SelfConjugate", but there are many notions of conjugateness which this intuitively encompasses. For instance, in §7 *Tensor Calculus, Applications, and Quasi-Spin* we shall describe "conjugate states" which are "conjugate-like" but are definitely not "Hermitian conjugates". Thus, we settle on this somewhat irregular usage of Hermitian conjugating elements of the complex and real numbers.

Finally, we can add the knowledge that we can resolve the Hermitian conjugate of anything which is `HermitianQ`.

```
In[113]:= HermitianConjugate /: (expr_?HermitianQ)^† := expr
```

Now conjugates work in the ways we are accustomed to.

```
In[114]:= (e^(i Ĥ t))^†

Out[114]= e^(-i t Ĥ)
```

There are a few final details we should include: (*i*) the conjugate of a bra is a ket and vice-versa, (*ii*) the creation and annihilation operators are Hermitian conjugates of each other, and (*iii*) the Hermitian conjugate of a commutator is just the reversed commutator of the conjugated arguments. These can simply be added as follows.

```
In[115]:= ⟨args__|^† := |args⟩;
          |args__⟩^† := ⟨args|;
          (a⁺)^† := a⁻;
          (a⁻)^† := a⁺;
          [x_, y_]_^† := [y^†, x^†]_;
```

Also note that we can determine the bra corresponding to a ket if the ket has a value.

```
In[120]:= Act↗ ⟨ψ_| := |ψ⟩^† /; ValueQ[|ψ⟩]
```

For illustration, we can easily explicitly show that the Hamiltonian is Hermitian (as well as X, P, etc., if we so choose).

```
In[121]:= (ω ħ (1/2 + a⁺ · a⁻))^†

Out[121]= ω ħ (1/2 + a⁺ · a⁻)
```

5.4.8 Algebraic Sums

To handle some of the upcoming calculations in a elegant way, it is convenient to be able to express abstract sums. One can standardly do this in *Mathematica* between indeterminate limits, but the functionality is somewhat limited. For instance, neither `Expand` nor `Simplify` works correctly on abstract sums, reindexing is not directly supported, etc. This is not to say that it is difficult to add such things; indeed we add them now. Let us reintroduce the notation for an algebraic sum from §2.2.2 *Notation: Examples*.

Technical Note: There exists a *Mathematica* package for handling abstract sums by Peltio[260], however it is not suitable for our needs. For the general theory behind symbolic summation, see for instance Petkovsek, Wilf & Zeilberger[262].

```
In[122]:= Notation[ Sum_indices__ sum_ <=> AlgebraicSum[sum_, indices__] ]
```

An algebraic sum is linear in its first argument but constant factoring is only allowed for objects not involved in the sum. Here is the generic prototype for this factoring.

```
In[123]:= AlgebraicSumFactors_Rules =
            { Generic_Env ↗ Sum_indices__ m_. c_ :↦ (c Sum_indices m) /; c ≠ 1 ∧ freeOfIndices[c, indices] };
```

Factoring terms outside of an algebraic sum is dependent on the factor being free of the summation indices.

```
In[124]:= freeOfIndices[c_, i_Symbol] := i ∉?{0,∞) c;
          freeOfIndices[c_, i_, indices___] /; i ∉?{0,∞) c :=
            freeOfIndices[c, indices]; freeOfIndices @ other___ = False;
```

In addition, algebraic sums distribute over non-commutative products. Let us encode this generically by creating a set of expansion rules for `AlgebraicSums`.

```
In[126]:= AlgebraicSumExpansion_Rules = {
            Generic_Env ↗ lhs___ . (Sum_indices__ arg_) . rhs___ :↦ Sum_indices lhs . arg . rhs,
            Generic_Env ↗ Sum_indices__ (expr : (_Plus _))_up :↦ Sum_indices Distribute[expr_u, Plus_u]}
          +{} (LZero_Rules +{} LDistribution_Rules /. {
            GenericPlus → Plus,
            Expand → Generic_Env,
            GenericL_Op → AlgebraicSum});
```

In addition, Kronecker deltas can reduce algebraic sums under the `Act` environment. (These rules are not totally general but sufficient for illustration.)

```
In[127]:= AlgebraicSumReduction_Rules =
  {Generic_Env ↗ (Sum_{l___, l_Symbol, r___} δ_{n_ m_} arg_. /; i ∈?∞ {n, m})_hp :↦
    With[{sol = Solve[n == m, i]}, (Sum_{l,r} arg /. sol[[1, 1]]) /;
      sol_len ≡ 1 ∧ sol ≠ {{}}]};
```

We also need some basic simplification rules for the Kronecker delta functions themselves. We define some auxiliary functions in the *Prototypes* package for `KroneckerDeltaReducibleQ` and `NewKroneckerVars`.

```
In[128]:= KroneckerReduction_Rules = {
    Generic_Env ↗ Tensor /: (δ_{n_ m_}^(n_?Positive))_hp :↦ δ_{n m},
    Generic_Env ↗ δ /: (δ_{n_ m_} /; KroneckerDeltaReducibleQ[n, m])_hp :↦
      With[{new = NewKroneckerSolutionQandA[n, m]}, new_[[2,1]] /;
        First @ new]};
```

We can also note that the Kronecker delta is non-complex.

```
In[129]:= DeclareHermitian @ (δ__)_hp
```

We can collect these rules into a general rule set.

```
In[130]:= AlgebraicSum_Rules :=
            AlgebraicSumReduction_Rules +{}
              AlgebraicSumFactors_Rules +{}
              KroneckerReduction_Rules
```

Now that we have created the generic prototypes, let us consequently selectively inherit the rule set behaviors to the environments `Act`, `Simplify`, and `Expand`.

```
In[131]:= Assign[AlgebraicSum_Rules /. Generic_Env → Simplify];
          Assign[
            AlgebraicSum_Rules +{} AlgebraicSumExpansion_Rules /. Generic_Env → Act];
          Assign[AlgebraicSumFactors_Rules +{} AlgebraicSumExpansion_Rules /.
              Generic_Env → Expand];

In[134]:= Act @ any_ := DynamicBar@ any
```

Lastly, before we progress further, we should implement some simple facts about algebraic sums. Any sum without summation indices is just the plain summand.

```
In[135]:= AlgebraicSum[arg_]_hp := arg
```

Also, the Hermitian conjugate of any sum is the sum of the conjugates.

In[136]:= $\left(\sum_{\text{Indices\_\_}} arg\_\right)^{\dagger}$:= $\left(\sum_{\text{Indices}} arg^{\dagger}\right)$

Finally, we are in a position to use abstract algebraic sums in our calculations. For instance, here is one such sum.

```
In[137]:= DeclareConstant[{i, j}]; DeclareHermitian[{i, j}]
```

In[138]:= $\sum_i \langle j_{\hat{H}} | \cdot \hat{X} \cdot | i_{\hat{H}} \rangle$ // Act

Out[138]= $\frac{\left(\sqrt{j} + \sqrt{1+j}\right)\sqrt{\frac{\hbar}{m\omega}}}{\sqrt{2}}$

Just to confirm that our sums are working correctly, it is instructive to expand out the braket without the summation.

In[139]:= $\langle j_{\hat{H}} | \cdot \hat{X} \cdot | i_{\hat{H}} \rangle$ // Act

Out[139]= $\frac{\sqrt{\frac{\hbar}{m\omega}}\left(\sqrt{1+i}\,\delta_{i,-1+j} + \sqrt{i}\,\delta_{i,1+j}\right)}{\sqrt{2}}$

It is evident by inspection that after summing over n, the answers given above agree. Now that the machinery is in place, calculations that would be arduous by hand are easy.

In[140]:= $\sum_i \langle j_{\hat{H}} | \cdot \hat{X}^4 \cdot | i_{\hat{H}} \rangle$

Out[140]= $\sum_i \langle j_{\hat{H}} | \cdot \hat{X}^4 \cdot | i_{\hat{H}} \rangle$

```
In[141]:= Act @ %
```

Out[141]=
$$\frac{1}{4\,m^2\,\omega^2}\Big(\big(3 + 6\sqrt{-1+j}\,\sqrt{j} + \sqrt{-3+j}\,\sqrt{-2+j}\,\sqrt{-1+j}\,\sqrt{j} + 4\,(-2+j)\,\sqrt{-1+j}\,\sqrt{j} + 12\,j + 6\,(-1+j)\,j + 6\sqrt{1+j}\,\sqrt{2+j} + 4\,j\,\sqrt{1+j}\,\sqrt{2+j} + \sqrt{1+j}\,\sqrt{2+j}\,\sqrt{3+j}\,\sqrt{4+j}\big)\,\hbar^2\Big)$$

The above point succinctly states why we need the developments in this thesis. The language modifications, notations, and generic structures have all interacted to allow us to perform the above calculations with great ease and expansibility. To truly appreciate why we develop such systems, the reader should verify the above calculation by hand. When would we want to calculate such things? In perturbation theory, as we will shortly see. Before this, in the next subsections, we present two operators that use algebraic sums: the resolution of identity and the propagator.

5.4.9 The Resolution of Identity

In this subsection we introduce the resolution of identity operator, while in the following subsection we introduce the propagator. These operators are easily expressed in terms of the structures we have built up and furthermore, they are active in our calculations. Throughout this subsection and the next, we assume that the reader has had some exposure to the concepts behind these operators. The operators are covered in most texts on quantum mechanics, for instance see any of [55, 68, 233, 292, etc.].

In quantum mechanical calculations it is sometimes necessary to insert the resolution of identity in order to calculate certain expressions. For instance, if we know the matrix elements $\langle m|A|n\rangle$ and $\langle n|B|k\rangle$, then we can calculate $\langle m|AB|n\rangle$ by inserting the resolution of identity operator between the A operator and the B operator. Typically in a physics text, this operator is denoted $\mathbb{I} = \sum_n |n\rangle\cdot\langle n|$ for a discrete set of basis states, while it is expressed as $\int_x |x\rangle\cdot\langle x|$ for a continuous set of basis states.

Technical Note: We could blur the distinction between sums and integrals and just use one notation for both as is sometimes done in some approaches. However, in this treatment we will maintain the distinction.

The sum or integral of the projection operators — the $P_u = |u\rangle\cdot\langle u|$ — form the identity only if the states form an orthonormal basis. To see this, consider the expansion of some state $|\psi\rangle$ in terms of an orthonormal basis $|u_i\rangle$, that is $|\psi\rangle = \sum_i c_i |u_i\rangle$. Because of the orthonormality of the basis, the c_j are determined by $\langle u_j \,|\, \psi\rangle = c_j$. Thus the expansion can be expressed as $|\psi\rangle = \sum_i \langle u_i \,|\, \psi\rangle |u_i\rangle$. Upon rearranging this becomes $|\psi\rangle = \sum_i |u_i\rangle\cdot\langle u_i|\cdot|\psi\rangle$. But since $|\psi\rangle$ was arbitrary, it must be the case that $\sum_i |u_i\rangle\cdot\langle u_i| = \mathbb{I}$.

Here is the resolution of the identity in the energy basis.

In[142]:=
$$\hat{\mathbb{I}}_{\hat{H}} := \sum_n |n_{\hat{H}}\rangle \cdot \langle n_{\hat{H}}|$$

We can verify that this acts as the "identity" in the following calculation. Here is a braket with the resolution of identity inserted.

In[143]:=
$$\langle x_{\hat{X}} | \cdot \hat{\mathbb{I}}_{\hat{H}} \cdot |m_{\hat{H}}\rangle$$

Out[143]=
$$\langle x_{\hat{X}} | \cdot \left(\sum_n |n_{\hat{H}}\rangle \cdot \langle n_{\hat{H}}| \right) \cdot |m_{\hat{H}}\rangle$$

In[144]:=
```
Expand @ %
```

Out[144]=
$$\sum_n \langle x_{\hat{X}} | \cdot |n_{\hat{H}}\rangle \cdot \langle n_{\hat{H}}| \cdot |m_{\hat{H}}\rangle$$

In the `Act` environment, the various rules we have inherited are sufficient to reduce this calculation.

In[145]:=
```
Act @ %
```

Out[145]= $\langle x_{\hat{x}} | \cdot | m_{\hat{H}} \rangle$

Thus we see that the resolution of the identity operator has acted exactly like the identity. As a check on our system and also for illustration purposes, it is instructive to take a given calculation, and compare the original results to the case when the resolution of identity is inserted.

In[146]:=
```
DeclareConstant[k]; DeclareHermitian[k];
```

In[147]:= $\langle m_{\hat{H}} | \cdot \hat{X} \cdot \mathbb{I}_{\hat{H}} \cdot \hat{P} \cdot | k_{\hat{H}} \rangle$

Out[147]= $\langle m_{\hat{H}} | \cdot \hat{X} \cdot \left(\sum_n | n_{\hat{H}} \rangle \cdot \langle n_{\hat{H}} | \right) \cdot \hat{P} \cdot | k_{\hat{H}} \rangle$

In[148]:=
```
Expand @ %
```

Out[148]= $\sum_n \langle m_{\hat{H}} | \cdot \hat{X} \cdot | n_{\hat{H}} \rangle \cdot \langle n_{\hat{H}} | \cdot \hat{P} \cdot | k_{\hat{H}} \rangle$

In[149]:=
```
Simplify @ Act @ %
```

Out[149]= $\frac{1}{2}\, i \sqrt{\frac{\hbar}{m\,\omega}}\, \sqrt{m\,\omega\,\hbar}\, \left(\sqrt{1+k}\, \sqrt{2+k}\, \delta_{k,-2+m} + \delta_{k,m} - \sqrt{-1+k}\, \sqrt{k}\, \delta_{k,2+m} \right)$

In[150]:=
```
⟨m_H| · X̂ · P̂ · |k_H⟩ // Act // Simplify
```

Out[150]= $\frac{1}{2}\, i \sqrt{\frac{\hbar}{m\,\omega}}\, \sqrt{m\,\omega\,\hbar}\, \left(\sqrt{1+k}\, \sqrt{2+k}\, \delta_{k,-2+m} + \delta_{k,m} - \sqrt{-1+k}\, \sqrt{k}\, \delta_{k,2+m} \right)$

In[151]:=
```
% == %%
```

Out[151]= True

Evidently, the results obtained in both cases are identical, as they should be.

5.4.10 The Propagator

The *propagator* $U(t)$ is somewhat similar in form to the resolution of identity operator. It can be expressed as a sum over projection operators times a phase involving each energy projection. The propagator is used to "propagate" an eigenstate forward or backward in time — for further details, again see any book on quantum mechanics. The propagator is easily expressed in the energy eigenbasis as follows.

$$U(t) = \sum_n |n\rangle \cdot \langle n| e^{-i \mathcal{E}_n t/\hbar} \tag{5.4.j}$$

Let us add this operator to our system.

In[152]:= $U[t\_] := \sum_n |n_{\hat{H}}\rangle \cdot \langle n_{\hat{H}}| \, e^{-i\,\mathcal{E}_n\, t/\hbar}$

We need to declare that the following symbols are constants with respect to non-commutative multiplication and also that they are Hermitian (non-complex).

```
In[153]:= DeclareConstant[{t, e, ℰ_, n}]
          DeclareHermitian[{t, e, ℰ_, n}]
```

Here is an example of the propagator acting on a simple eigenstate of energy.

In[155]:= $U[t] \cdot |m_{\hat{H}}\rangle$

Out[155]= $\left(\sum_n e^{-\frac{i t \mathcal{E}_n}{\hbar}} |n_{\hat{H}}\rangle \cdot \langle n_{\hat{H}}|\right) \cdot |m_{\hat{H}}\rangle$

We can expand this algebraic sum and act on the state.

```
In[156]:= Act @ %
```

Out[156]= $e^{-\frac{i t \mathcal{E}_m}{\hbar}} |m_{\hat{H}}\rangle$

This is how the energy eigenstates vary with time. The energy eigenstates are called "stationary states" since the probability distribution $P(\phi)$ for an eigenket ϕ of an operator Ω is time independent in an energy eigenstate. That is, $P(\phi, t) = |\langle \phi | n_H(t) \rangle|^2 = |\langle \phi | n_H \rangle|^2$. Let us show this.

```
In[157]:= Notation[|expr_|^2 ⟺ ModSquared[expr_]]
```

```
In[158]:= |a_|^2 := a† · a
```

In[159]:= $|\langle \phi | \cdot U[t] \cdot |m_{\hat{H}}\rangle|^2 == |\langle \phi | \cdot |m_{\hat{H}}\rangle|^2$ // Act

Out[159]= True

For the next example, consider the following wavefunction, which is a superposition of the energy eigenstates.

In[160]:= $|\psi\rangle = \frac{|0_{\hat{H}}\rangle}{2} + \frac{|1_{\hat{H}}\rangle}{2} + \frac{|2_{\hat{H}}\rangle}{\sqrt{2}}$

Out[160]= $\frac{|0_{\hat{H}}\rangle}{2} + \frac{|1_{\hat{H}}\rangle}{2} + \frac{|2_{\hat{H}}\rangle}{\sqrt{2}}$

Let us now check that the state $|\psi\rangle$ is normalized.

In[161]:= $\langle \psi | \cdot |\psi\rangle$ // Act

Out[161]= 1

By acting on the initial state with the propagator, we obtain the evolved state at the future time t.

In[162]:= $|\psi[t]\rangle = U[t] \cdot |\psi\rangle$ // Act

Out[162]= $\frac{1}{2} e^{-\frac{i t \mathcal{E}_0}{\hbar}} \left| 0_{\hat{H}} \right\rangle + \frac{1}{2} e^{-\frac{i t \mathcal{E}_1}{\hbar}} \left| 1_{\hat{H}} \right\rangle + \frac{e^{-\frac{i t \mathcal{E}_2}{\hbar}} \left| 2_{\hat{H}} \right\rangle}{\sqrt{2}}$

The state at time t remains normalized.

In[163]:= `⟨ψ[t]| · |ψ[t]⟩ // Act`

Out[163]= 1

For what follows, it is convenient to directly specify the energy levels of the harmonic oscillator.

In[164]:= $\mathcal{E}_{n\_} := \hbar\, \omega \left(n + \frac{1}{2}\right)$

Let us calculate the expectation value of the position operator, X, at time t.

In[165]:= `⟨ψ[t]| · X̂ · |ψ[t]⟩ // Act // ExpToTrig // Simplify`

Out[165]= $\frac{3 \sqrt{\frac{\hbar}{m\, \omega}}\, \text{Cos}[t\, \omega]}{2 \sqrt{2}}$

First, the expectation is real, as it should be since X is an observable. It is also evident that the expectation value of the position undergoes simple harmonic oscillation. As it should. Moreover, the average momentum is out of phase with respect to the position, as we see from the following.

In[166]:= `⟨ψ[t]| · P̂ · |ψ[t]⟩ // Act // ExpToTrig // Simplify`

Out[166]= $-\frac{3 \sqrt{m\, \omega\, \hbar}\, \text{Sin}[t\, \omega]}{2 \sqrt{2}}$

Let us re-enter the Hamiltonian — either in terms of creation and annihilation operators, or in terms of the position and momentum operators (it will be handled in either case.)

In[167]:= $\hat{H} = \left(a^{+} \cdot a^{-} + \frac{1}{2}\right) \hbar\, \omega;$

We can explicitly reconfirm that the energy in the oscillator is time-independent.

In[168]:= `⟨ψ[t]| · Ĥ · |ψ[t]⟩ // Act`

Out[168]= $\frac{7\, \omega\, \hbar}{4}$

In[169]:= `⟨ψ| · Ĥ · |ψ⟩ // Act`

Out[169]= $\frac{7\, \omega\, \hbar}{4}$

The expectation of the potential energy oscillates at twice the fundamental frequency.

In[170]:= `⟨ψ[t]| · X̂^2 · |ψ[t]⟩ // Act // ExpToTrig // Simplify`

Out[170]= $\frac{\hbar\, (7 + 2\, \text{Cos}[2\, t\, \omega])}{4\, m\, \omega}$

This is consistent with the classical behavior since if $x \propto \cos(\omega\, t)$, then $x^2 \propto \cos(\omega\, t)^2 = \frac{1}{2}(\cos(2\, t\, \omega) + 1)$. It is instructive to note that for this particular wave function, all higher powers of X scale at the fundamental frequency or twice the fundamental frequency, that is, there are no other harmonics.

In[171]:=
```
⟨ψ[t] | · X̂^4 · |ψ[t]⟩ // Act // ExpToTrig // Simplify
```

Out[171]=
$$\frac{3\,\hbar^2\,(4 + \text{Cos}[2\,t\,\omega])}{2\,m^2\,\omega^2}$$

However, this is not always true. Here it would be nice to transform to the Heisenberg picture and calculate $U(t)^\dagger\, X^4\, U(t)$ — for further details see [68, 280, 292, 334, etc.]. Yet, we have not scoped our summation variables in our abstract sums correctly. We could do this all automatically, and indeed we should as we will for tensors; but for this one example, we will make an exception since it would be a minor diversion to create such behavior here. Instead, let us manually specify the summation variable in our propagators as follows.

In[172]:=
```
U[t_, n_]_Ĥ := Σ_n |n_Ĥ⟩ · ⟨n_Ĥ| e^(-i ε_n t/ħ)
```

Using this we can calculate the operator in the Heisenberg picture.

In[173]:=
```
U[t, k]† · X̂^3 · U[t, n] // Act // Simplify // Expand
```

Out[173]=
$$\frac{e^{3\,i\,t\,\omega}\,\hbar\,\sqrt{\frac{\hbar}{m\,\omega}}\,\sum_k \sqrt{-2+k}\,\sqrt{-1+k}\,\sqrt{k}\,\left|k_{\hat{H}}\right\rangle \cdot \left\langle (-3+k)_{\hat{H}}\right|}{2\sqrt{2}\,m\,\omega} +$$
$$\frac{3\,e^{i\,t\,\omega}\,\hbar\,\sqrt{\frac{\hbar}{m\,\omega}}\,\sum_k k^{3/2}\,\left|k_{\hat{H}}\right\rangle \cdot \left\langle (-1+k)_{\hat{H}}\right|}{2\sqrt{2}\,m\,\omega} +$$
$$\frac{3\,e^{-i\,t\,\omega}\,\hbar\,\sqrt{\frac{\hbar}{m\,\omega}}\,\sum_k \sqrt{1+k}\,\left|k_{\hat{H}}\right\rangle \cdot \left\langle (1+k)_{\hat{H}}\right|}{2\sqrt{2}\,m\,\omega} +$$
$$\frac{3\,e^{-i\,t\,\omega}\,\hbar\,\sqrt{\frac{\hbar}{m\,\omega}}\,\sum_k k\,\sqrt{1+k}\,\left|k_{\hat{H}}\right\rangle \cdot \left\langle (1+k)_{\hat{H}}\right|}{2\sqrt{2}\,m\,\omega} +$$
$$\frac{e^{-3\,i\,t\,\omega}\,\hbar\,\sqrt{\frac{\hbar}{m\,\omega}}\,\sum_k \sqrt{1+k}\,\sqrt{2+k}\,\sqrt{3+k}\,\left|k_{\hat{H}}\right\rangle \cdot \left\langle (3+k)_{\hat{H}}\right|}{2\sqrt{2}\,m\,\omega}$$

We can see from the form of the output, we are obtaining both a $\cos(\omega\, t)$ term and a $\cos(3\,\omega\, t)$ term. Finally, let us quickly repeat the calculation, but this time for two arbitrary eigenstates.

In[174]:=
```
|ψ⟩ = |i_Ĥ⟩/√2 + |j_Ĥ⟩/√2
```

Out[174]=
$$\frac{\left|i_{\hat{H}}\right\rangle}{\sqrt{2}} + \frac{\left|j_{\hat{H}}\right\rangle}{\sqrt{2}}$$

The wave function $|\psi\rangle$ is normalized if $i \neq j$.

In[175]:=
```
⟨ψ| · |ψ⟩ // Act
```

```
Out[175]= 1 + δ_{i j}
```

```
In[176]:= |ψ[t]⟩ = U[t] · |ψ⟩ // Act;
```

This state remains normalized if $i \neq j$.

```
In[177]:= ⟨ψ[t]| · |ψ[t]⟩ // Act // ExpToTrig
```

```
Out[177]= 1 + Cos[(1/2 + i) t ω - (1/2 + j) t ω] δ_{i j}
```

The expectation of the position always oscillates at the fundamental frequency, as we can see from the following.

```
In[178]:= ⟨ψ[t]| · X̂ · |ψ[t]⟩ // Act // ExpToTrig // Simplify
```

```
Out[178]= 1/(2 √2) (√(ħ/(m ω)) ((((√(1 + i) + √j) Cos[(i - j) t ω] -
                                  i (√(1 + i) - √j) Sin[(i - j) t ω]) δ_{i -1+j} +
                                 ((√i + √(1 + j)) Cos[(i - j) t ω] +
                                  i (-√i + √(1 + j)) Sin[(i - j) t ω]) δ_{i 1+j}))
```

Technical Note: The above example illustrates that there are further simplifications we could implement. Specifically, if the Kronecker delta is 1, then the coefficient of the respective `Sin` term must be 0, thus the expression overall simplifies to just the `Cos` terms.

There are many other calculations we could perform in this same style, for instance, see Cohen-Tannoudji[68]. To show a slightly different relation, consider the Thomas-Reiche-Kuhn rule — see Shankar[292], etc. This rule states that the eigenstates of any Hamiltonian of the form $H = P^2/2m + V(X)$ obey the following rule.

$$\sum_{n'} (\mathcal{E}_{n'} - \mathcal{E}_n) |\langle n'_H | \cdot X \cdot | n_H \rangle|^2 = \frac{\hbar^2}{2m}$$

Let us verify this for our harmonic oscillator.

```
In[179]:= Symbolize[n'];
```

```
In[180]:= DeclareConstant[n']
```

```
In[181]:= Σ_{n'} (ℰ_{n'} - ℰ_n) |⟨n'_ĥ| · X̂ · |n_ĥ⟩|^2 // Act // Expand
```

```
Out[181]= ħ^2/(2 m)
```

Here is a higher order variant of this rule.

```
In[182]:= Σ_{n'} (ℰ_{n'} - ℰ_n) |⟨n'_ĥ| · X̂^3 · |n_ĥ⟩|^2 // Act // Simplify
```

```
Out[182]= 27 (1 + 2 n + 2 n^2) ħ^4 / (8 m^3 ω^2)
```

5.4.11 Simple Time-Independent Perturbation Theory

Let us now turn to some simple time-independent non-degenerate perturbation theory. Assume that for a Hamiltonian H, we know the eigenstates $|n_H\rangle$. The question we then seek to answer is the following. If we add a "small" perturbing Hamiltonian W to H, how will our energies change, and also how will our eigenstates change? This question is answered in most quantum mechanics texts, and in principle is not overly complex — see Shankar[292], Cohen-Tannoudji[68], etc. Essentially, one postulates that both the eigenvalues and the eigenkets of our total Hamiltonian can be expanded in a power series. This series is then solved to successive orders.

The results, which we will only quote, are as follows. We denote the total Hamiltonian by $H' = H + W$. Additionally, we will let $|n^i\rangle$ and $\mathcal{E}_n^i$ denote the successive series of eigenstates and their corresponding energies, where i labels the stage of the approximation. That is, to zero$^{\text{th}}$ order, the n^{th} eigenstate is the unperturbed eigenstate $|n^0\rangle$, and its energy is the unperturbed energy $\mathcal{E}_n^0$. To first order, the corrections to the energy and eigenstates turn out to be as follows.

$$\begin{aligned} \mathcal{E}_n^1 &= \langle n^0 | \cdot W \cdot | n^0 \rangle \\ |n\rangle &= |n^0\rangle + |n^1\rangle = |n^0\rangle + \sum_{m \neq n} \frac{|m^0\rangle \langle m^0 | \cdot W \cdot | n^0 \rangle}{\mathcal{E}_n^0 - \mathcal{E}_m^0} \end{aligned} \tag{5.4.k}$$

The corrections to the energy eigenstates to second order are as follows.

$$\mathcal{E}_n^2 = \sum_{m \neq n} \frac{|\langle n^0 | \cdot W \cdot | m^0 \rangle|^2}{\mathcal{E}_n^0 - \mathcal{E}_m^0} \tag{5.4.l}$$

The only adaptation we must make is to handle the case in the algebraic summations when $m \neq n$. For this we must modify the reduction rules for our algebraic sums.

```
In[183]:= AlgebraicSumReduction_Rules = AlgebraicSumReduction_Rules ∪ {
  Generic_Env ↗ ( Σ_{ℓ___, i_Symbol≠j_Symbol, r___} δ_{n_ m_} arg_. /; i ∈_?∞ {n, m} ) :>
    With[{sol = Solve[n == m, i]}, Σ_{ℓ,r} solution_{i≠j}[sol[[1,1]], arg] /;
      (sol_len ≡ 1 ∧ sol ≠ {{}})]};
```

The additional rule for sums, which exclude a specific index, relies on the following simple auxiliary routine.

```
In[184]:= solution_{i_≠j_}[i_ → r_, arg_] := If[r == j, 0, arg /. i → r, arg /. i → r]
          solution_True[i_ → r_, arg_] := arg /. i → r
          solution_False[i_ → r_, arg_] := 0
```

In addition, we must update the criterion under which expressions can be factored out of sums.

```
In[187]:= freeOfIndices[c_, i_Symbol ≠ n_] := i ∉_?^{0,∞} c;
          freeOfIndices[c_, i_Symbol ≠ n_, indices___] /; i ∉_?^{0,∞} c :=
            freeOfIndices[c, indices];
```

To incorporate the change to algebraic sums, we first unassign the previous rules for handling algebraic sums, and then inherit the new generic prototypes. This is yet another instance of where it would be nice to have rule precedences as described in §4.8.1 *Potential Further Developments*.

```
In[189]:= UnAssign[AlgebraicSumFactors_Rules /. Generic_Env → Simplify];
          UnAssign[
            AlgebraicSum_Rules +_{} AlgebraicSumExpansion_Rules /. Generic_Env → Act];
          UnAssign[AlgebraicSumFactors_Rules +_{} AlgebraicSumExpansion_Rules /.
              Generic_Env → Expand];
```

Technical Note: It is important that the rule for factoring terms out of a summation be put after the rule for reducing the sum. The reason is due to "flat matching" on `Times`. That is, the "pattern matcher" tries every possible combination of "`Times` objects" to factor out. For instance, if the coefficient inside the sum is something like $j\sqrt{1+j}\sqrt{2+j}\sqrt{3+j}\sqrt{4+j}$ and the summation variable is `j`, then the pattern matcher will have to try 5! different combinations. Actually, in the coming calculations we encounter terms with 8 factors which would lead to 40320 different attempts to factor a term out of the sum. Thus it is quite important to reduce the sum before this, if possible.

```
In[192]:= Assign[AlgebraicSum_Rules /. Generic_Env → Simplify];
          Assign[AlgebraicSum_Rules +_{} AlgebraicSumExpansion_Rules /.
              Generic_Env → Act];
          Assign[AlgebraicSumFactors_Rules +_{} AlgebraicSumExpansion_Rules /.
              Generic_Env → Expand];
```

Let us test the modification to algebraic sums on a simple bra-ket. Because X connects energy eigenstates of levels differing by exactly one, we would expect the two sides of the following equation to be equal.

```
In[195]:= Sum_{i≠n} ⟨i_Â | · X̂ · |n_Â⟩ == Sum_i ⟨i_Â | · X̂ · |n_Â⟩ // Act // Simplify
Out[195]= True
```

In contrast, since the following braket is only non-zero for $i = n$, we expect the answer to be zero.

```
In[196]:= Sum_{i≠n} ⟨i_Â | · ω · a⁺ · a⁻ · |n_Â⟩ // Act
Out[196]= 0
```

With this background, let us assume we are working with the following perturbed Hamiltonian system.

$$H = \mathcal{H}(x \to X, p \to P) = \frac{P^2}{2m} + \frac{m\omega^2 X^2}{2} + \frac{\lambda m^2 \omega^3 X^4}{\hbar} \tag{5.4.m}$$

(The constant factor $m^2\omega^3/\hbar$, which multiplies λX^4 in (5.4.m), has been suitably chosen so that our energy perturbations occur in units of $\omega\hbar$.) The perturbation of the energies to first order is easily calculated. First, we declare that the parameter λ is constant and non-complex, and then we expand the perturbing potential.

```
In[197]:= DeclareConstant[λ]; DeclareHermitian[λ];
          Ŵ = λ m^2 ω^3 X̂^4 / ħ · 1 // Act;
```

We can now directly apply (5.4.k) in order to determine the energy changes to first order.

```
In[199]:= ⟨n_Ĥ| · Ŵ · |n_Ĥ⟩ // Act // Simplify
Out[199]= 3/4 (1 + 2 n + 2 n^2) λ ω ħ
```

Also, by (5.4.k), our first adjusted eigenkets are calculated as follows. (We suitably collect terms in order to make the output more readable.)

```
In[200]:= |n'_Ĥ⟩ = |n_Ĥ⟩ + Σ_{i≠n} |i_Ĥ⟩ · ⟨i_Ĥ| · Ŵ · |n_Ĥ⟩ / (ε_n - ε_i) //
                Act // Collect[#, |_⟩, Simplify] &
Out[200]= 1/16 √(-3 + n) √(-2 + n) √(-1 + n) √n λ |(-4 + n)_Ĥ⟩ +
          1/4 √(-1 + n) √n (-1 + 2 n) λ |(-2 + n)_Ĥ⟩ + |n_Ĥ⟩ -
          1/4 √(1 + n) √(2 + n) (3 + 2 n) λ |(2 + n)_Ĥ⟩ -
          1/16 √(1 + n) √(2 + n) √(3 + n) √(4 + n) λ |(4 + n)_Ĥ⟩
```

Our first order approximate eigenstates should be normalized to first order. That is, the normalization can only differ from 1 in terms with coefficients of λ^2 or higher.

```
In[201]:= ⟨n'_Ĥ| · |n'_Ĥ⟩ // Act // Collect[#, λ, Simplify] &
Out[201]= 1 + 1/128 n (156 + 422 n + 487 n^2 + 130 n^3 + 65 n^4) λ^2
```

Thus it is evident, just as the underlying theory guaranteed, that the states $|n'_H\rangle$ are normalized to first order. Similarly, we can calculate the energy corrections to second order according to (5.4.l).

```
In[202]:= ε_n + ⟨n_Ĥ| · Ŵ · |n_Ĥ⟩ + Σ_{i≠n} |⟨n_Ĥ| · Ŵ · |i_Ĥ⟩|^2 / (ε_n - ε_i) // Act //
                Collect[#, λ, Simplify] & // Timing
```

Out[202]= $\left\{3.41667\ \text{Second},\ \left(\frac{1}{2}+n\right)\omega\hbar+\frac{3}{4}\left(1+2n+2n^2\right)\lambda\omega\hbar-\frac{1}{8}\left(21+59n+51n^2+34n^3\right)\lambda^2\omega\hbar\right\}$

These calculations agree with those given in texts — see Shankar[292] and Cohen-Tannoudji[68]. We have made no effort to obtain computational efficiency at this stage; rather we have attempted to present the calculations in a clear and concise manner. However, there are obvious simplifications that could be made. For instance, we could calculate $\langle n_H|W|i_H\rangle$ and then just conjugate this result. With a minimal amount of effort we can store these intermediate computations as we progress, and thereby reduce the time taken to compute such expansions.

In[203]:=
```
Timing[
  K̂ = ⟨n_ß | · Ŵ · | i_ß⟩ // Act;
  F̂ = K̂^† · K̂ // Expand // Act;
  ε_n + ⟨n_ß | · Ŵ · | n_ß⟩ + Σ_{i≠n} F̂/(ε_n - ε_i) //
        Act // Collect[#, λ, Simplify] &]
```

Out[203]= $\left\{0.95\ \text{Second},\ \left(\frac{1}{2}+n\right)\omega\hbar+\frac{3}{4}\left(1+2n+2n^2\right)\lambda\omega\hbar-\frac{1}{8}\left(21+59n+51n^2+34n^3\right)\lambda^2\omega\hbar\right\}$

We could progress further to higher energy levels, for instance see[108]. Performing such calculations would be relatively trivial with the current machinery we have developed. However, the above is more than sufficient for illustration purposes.

5.4.12 Concluding Remarks

In this past section we have used the simple example of the quantum harmonic oscillator to expose our notations, language modifications, and inheritance paradigm. We have treated most of the algebraic portions of the problem in detail. We could have jumped straight to the raising and lowering operators, but then we would not have exposed all of our different operations, how they work, how they are inter-related, etc.

Our inheritance paradigm has been used extensively throughout this last section. We have observed it working in an elegant manner to allow the creation of generic structures for non-commutative multiplication, as well as generic raising and lowering operations. These generic operations have been instantiated in specific environments and functions. Even though we explicitly used the `Act` environment to accomplish most of our "actions", we could of course have chosen any particular split of functionality into different environments. Indeed, this is the whole focus of the inheritance paradigm.

Despite the fact that the topics covered are not at a sophisticated level of quantum mechanics, they are definitely non-trivial and sufficient to demonstrate the inheritance paradigm that we have developed. Indeed, from the prospective of computer algebra, it is impressive that we can

perform them in such a faithful manner. The calculations are extremely elegant and proceed in the way a physicist would expect and desire. Towards the end of this last subsection we were starting to perform calculations that would be semi-tedious by hand. Thus it is desirable to automate these kinds of calculations where possible. Of course, we could explore many other examples in a very similar vein to the above calculations, for instance, two coupled oscillators, etc. Instead, let us progress on to another commonly treated example that is sufficiently different from the quantum harmonic oscillator.

5.5 Example: Angular Momentum

In this section we continue with the development of our structures through examples in physics. We are endeavoring to illustrate the capabilities of the packages we have developed in this thesis, notably the *Notation* package and more importantly the *Assign* package along with its attendant language modifications. Concurrently, we are developing further prototypical structures for expressing properties of our overall system. Therefore, let us continue our exploration of these packages via illustrations from quantum mechanics. The focus is more on an operator based approach than that given in the previous subsection. Specifically, the topic of this section is an example that all physicists should be familiar with: angular momentum.

5.5.1 Primitive Cartesian Angular Momenta

In this subsection we introduce primitive forms of certain operators. These primitive forms will be used in the investigation of the commutation relations of angular momentum as well as several other related issues.

Angular momentum obeys the condensed commutation relations proscribed by the following.

$$[J_i, J_j]_- = i\epsilon_{ijk} J_k \tag{5.5.a}$$

The standard definitions of the "angular momentum like" relations embodied in (5.5.a) can easily be derived from the more basic definitions of angular momentum in terms of Cartesian components and their partial derivatives. As an illustrative exercise, let us do this.

We need to create an operator for the distinct purpose of expressing the Cartesian angular momentum operators, that is $\hat{\mathbf{L}}_x$, $\hat{\mathbf{L}}_y$, $\hat{\mathbf{L}}_z$, in terms of the primitive Cartesian variables and the partial derivatives of such variables. We will call this operator `PrimitiveForm`. One should be aware that the style of the calculations and the form of the results, using the `PrimitiveForm` operator, are usually on a level lower than that at which we would typically perform computations. That is, normally we will perform computations using the abstracted operators themselves. For instance, in the previous section, §5.4 *Example: The Harmonic Oscillator*, our calculations were fundamentally based on creation and annihilation operators. Despite this, it

is convenient to be able to defer, when necessary, to the most basic forms to confirm certain relations, manipulate expressions, etc.

Here is how the Cartesian angular momentum operators are expressed in terms of the primitive Cartesian variables and their partial derivatives. These relations correspond to expressing $L = -i\, r \times \nabla$.

```
In[1]:= PrimitiveForm ↗ L̂_x := -i (y · ∂̂_z - z · ∂̂_y)

        PrimitiveForm ↗ L̂_y := -i (z · ∂̂_x - x · ∂̂_z)

        PrimitiveForm ↗ L̂_z := -i (x · ∂̂_y - y · ∂̂_x)
```

For reference, consult almost any text on quantum mechanics — for instance, Cassels[55], Cohen-Tannoudji[68], Lindgren & Morrison[206], Shankar[292], Thompson[311], Weissbluth [334], etc.

Technical Note: There is a small discrepancy amongst various authors as to whether or not to include a factor of $\hbar$ in the components of the angular momenta. We follow Weissbluth, Thompson, and Lindgren & Morrison with the above definition, as opposed to that of Shankar and Cohen-Tannoudji, which include a factor of $\hbar$.

In addition, we want our `PrimitiveForm` operation to also be of an expansion type — we are attempting to obtain as primitive a form as possible. Thus we make `PrimitiveForm` inherit the appropriately selected rules of `Expand`.

```
In[4]:= Assign[Select[Values @ Expand,
                (NonCommutativeTimes | Commutator) ∈?∞ rule &rule] /.
            Expand → PrimitiveForm]
```

Further to the above definitions, it is again convenient to enforce the fact that `PrimitiveForm` does not extend outside its context. This was explained in §4.7.6 *Caveats about Dynamic Rules I: Renegade Environmental Rules.*

```
In[5]:= PrimitiveForm @ any_ := DynamicBar @ any
```

Recall the normal ordering for creation and annihilation operators in §5.4.4 *Creation and Annihilation Operators*. In a similar manner we can "normal order" the partial derivative operators. But, prior to this normal ordering, let us introduce the concept of a "semicommutative multiplier".

At certain stages we encounter substructures whose "nature" lies somewhere between constants and operators. For example, `x` and `y` are clearly not operators, yet they are not free to commute with other structures, for instance the components of Cartesian momenta or angular momenta, etc. However, they are free to commute amongst themselves. For lack of a better designation, we shall call symbols or structures which obey this nebulous description, *semicommutative multipliers* (or `$CMultiplier`). We must be able to identify when an object is a semicommutative multiplier and also how to handle such a structure via prototypical rules. Fortuitously, such "semicommutative-ness" can be directly inherited from the prototype for a constant.

```
In[6]:= Assign[GenericConstantQ_Rules /. {
              GenericConstQ → $CMultiplierQ,
              GenericTimes → Times,
              GenericPlus → Plus,
              DeclareGenericConstant → Declare$CMultiplier,
              DeclareGenericUnconstant → DeclareNon$CMultiplier}]

In[7]:= Declare$CMultiplier[{x, y, z}]
```

Along with the inherited "constant-like" behavior, we can simplify non-commutative products of semicommutative multipliers, to commutative products of semicommutative multipliers. This rule can be prototypically given as follows.

```
In[8]:= GenericSCMultiplierSimplification_Rules = {
            Generic_Env ↗ a_?$CMultiplierQ · b_?$CMultiplierQ :≐ a b};
```

As is by now standard, we can instantiate these generic rules to specific environments, namely `Simplify` and `PrimitiveForm`.

```
In[9]:= Assign[GenericSCMultiplierSimplification_Rules /. Generic_Env → Simplify]
        Assign[GenericSCMultiplierSimplification_Rules /. Generic_Env → PrimitiveForm]
```

Here are two examples that test the functionality of semicommutative multipliers.

```
In[11]:= $CMultiplierQ[√(x² y + 1 + e)]

Out[11]= True

In[12]:= x · y · A · y · √(x² y + 1) // Simplify

Out[12]= (x y) · A · (y √(1 + x² y))
```

This last example shows that we can simplify expressions involving semicommutative multipliers into a form that is more familiar to physicists. Unfortunately, the expansion rules we previously gave in §5.3.8 *Non-Commutative Times III* expand out powers in non-commutative objects, yet we have just specified a rule which will result in the collection of powers within the `PrimitiveForm` environment. Therefore, we need to override our power expansion rule.

```
In[13]:= UnAssign[GenericPowerExpansion_Rules /. {
              Generic_Env → PrimitiveForm,
              Generic_Op → NonCommutativeTimes}]
```

Here is our new generic expansion that works safely with semicommutative multipliers.

```
In[14]:= GenericSafePowerExpansion_Rules = {Generic_Env ↗
           Generic_Op[a___, b_^m_Integer?Positive /; ¬ $CMultiplierQ[b], c___]_tp :↦
                (Generic_Op[a, ##, c] & @@ Table[b, {m}])};

In[15]:= Assign[GenericSafePowerExpansion_Rules /. {
              Generic_Env → PrimitiveForm,
              Generic_Op → NonCommutativeTimes}]
```

The PrimitiveForm environment operates correctly with semicommutative multipliers.

```
In[16]:= x · x · A · y · √(x² y + 1) // PrimitiveForm
Out[16]= x² · A · (y √(1 + x² y))
```

We should also include the fact that powers of semicommutative multiplier expressions can simplify from within a single non-commutative times wrapper.

```
In[17]:= (a_ /; Head@ a_u ≠ Power)_nc =.
In[18]:= (a_ /; Head@ a_u ≠ Power ∨ $CMultiplierQ @ a)_nc := a
```

5.5.2 Primitive Commutation Relations of Cartesian Angular Momenta

With the addition of semicommutative multipliers, let us return to derivative normal ordering. We can specify that all derivatives commute, but we want them to commute into a "normal order". That is, we should only swap two adjacent partial derivative operators if the swap would make the resulting overall product lexically smaller.

```
In[19]:= DerivativeNormalOrdering ↗ (∂̂_b_)^n_. · (∂̂_a_)^m_. :=a (∂̂_a)^n · (∂̂_b)^m /; a ≺lex b
```

In addition, all partial derivative operators act on semicommutative multipliers by differentation.

```
In[20]:= DerivativeNormalOrdering ↗ ∂̂_v_ · f_?$CMultiplierQ :=a (∂_v f + f · ∂̂_v)
```

As was the case in previous environments, we need our normal ordering on derivatives to inherit the expansion and simplification rules of a generic multilinear operator.

```
In[21]:= Assign [ GenericML_OpRules /. {
                    GenericML_Op → NonCommutativeTimes,
                    Expand → DerivativeNormalOrdering,
                    Simplify → DerivativeNormalOrdering}]
```

Finally, we bar the environmental rules from escaping their context.

```
In[22]:= DerivativeNormalOrdering@ any_ := DynamicBar @ any
```

We are now in a position to explicitly verify the angular momentum commutation relations (5.5.a).

```
In[23]:= [L̂_x, L̂_y]_ == i L̂_z
Out[23]= [L̂_x, L̂_y]_ == i L̂_z
```

For illustration, remember that commutators were expanded by Expand (although PrimitiveForm now also has these rules).

```
In[24]:= Expand @ %
```

$$\text{Out[24]=}\quad \hat{\mathbf{L}}_x \cdot \hat{\mathbf{L}}_y - \hat{\mathbf{L}}_y \cdot \hat{\mathbf{L}}_x == \mathrm{i}\,\hat{\mathbf{L}}_z$$

Now, by acting with PrimitiveForm, we expand the operators into their primitive constituents.

```
In[25]:= % // PrimitiveForm
```

$$\text{Out[25]=}\quad -\left(x \cdot \hat{\partial}_z \cdot y \cdot \hat{\partial}_z\right) + x \cdot \hat{\partial}_z \cdot z \cdot \hat{\partial}_y + y \cdot \hat{\partial}_z \cdot x \cdot \hat{\partial}_z - y \cdot \hat{\partial}_z \cdot z \cdot \hat{\partial}_x + z \cdot \hat{\partial}_x \cdot y \cdot \hat{\partial}_z -$$
$$z \cdot \hat{\partial}_x \cdot z \cdot \hat{\partial}_y - z \cdot \hat{\partial}_y \cdot x \cdot \hat{\partial}_z + z \cdot \hat{\partial}_y \cdot z \cdot \hat{\partial}_x == x \cdot \hat{\partial}_y - y \cdot \hat{\partial}_x$$

By normal ordering the derivatives, we effect a reduction.

```
In[26]:= % // DerivativeNormalOrdering
```

$$\text{Out[26]=}\quad x \cdot \left(1 + z \cdot \hat{\partial}_z\right) \cdot \hat{\partial}_y - y \cdot \left(1 + z \cdot \hat{\partial}_z\right) \cdot \hat{\partial}_x - x \cdot y \cdot \hat{\partial}_z \cdot \hat{\partial}_z +$$
$$y \cdot x \cdot \hat{\partial}_z \cdot \hat{\partial}_z - z \cdot x \cdot \hat{\partial}_y \cdot \hat{\partial}_z + z \cdot y \cdot \hat{\partial}_x \cdot \hat{\partial}_z == x \cdot \hat{\partial}_y - y \cdot \hat{\partial}_x$$

By expanding the resulting expression, we obtain a more normal form.

```
In[27]:= % // Expand
```

$$\text{Out[27]=}\quad x \cdot \hat{\partial}_y - y \cdot \hat{\partial}_x - x \cdot y \cdot \hat{\partial}_z \cdot \hat{\partial}_z + x \cdot z \cdot \hat{\partial}_z \cdot \hat{\partial}_y + y \cdot x \cdot \hat{\partial}_z \cdot \hat{\partial}_z -$$
$$y \cdot z \cdot \hat{\partial}_z \cdot \hat{\partial}_x - z \cdot x \cdot \hat{\partial}_y \cdot \hat{\partial}_z + z \cdot y \cdot \hat{\partial}_x \cdot \hat{\partial}_z == x \cdot \hat{\partial}_y - y \cdot \hat{\partial}_x$$

By simplifying this expression, the semicommutative multiplier constituents are grouped together.

```
In[28]:= Simplify @ %
```

$$\text{Out[28]=}\quad (x\,z) \cdot \hat{\partial}_z \cdot \hat{\partial}_y + (y\,z) \cdot \hat{\partial}_x \cdot \hat{\partial}_z == (x\,z) \cdot \hat{\partial}_y \cdot \hat{\partial}_z + (y\,z) \cdot \hat{\partial}_z \cdot \hat{\partial}_x$$

And finally, normal ordering the derivatives will yield the desired reduction to True.

```
In[29]:= % // DerivativeNormalOrdering

Out[29]= True
```

As we can see, this process was somewhat laborious. It is more optimal to have PrimitiveForm inherit the rules of DerivativeNormalOrdering.

```
In[30]:= Assign[Values @ DerivativeNormalOrdering /.
           DerivativeNormalOrdering → PrimitiveForm]
```

Now we can verify the commutation relations in a simple manner.

```
In[31]:= [L̂x, L̂y]₋ == i L̂z // PrimitiveForm // Simplify
Out[31]= True
```

Similarly, we can check the commutation relations of the other variants of (5.5.a).

```
In[32]:= [L̂y, L̂z]₋ == i L̂x // PrimitiveForm // Simplify
Out[32]= True

In[33]:= [L̂z, L̂x]₋ == i L̂y // PrimitiveForm // Simplify
Out[33]= True
```

Although it should be obvious that the opposites of the above statements are true, it is nice to explicitly check them.

```
In[34]:= {[L̂y, L̂x]₋ == -i L̂z, [L̂z, L̂y]₋ == -i L̂x, [L̂x, L̂z]₋ == -i L̂y} //
           PrimitiveForm // Simplify
Out[34]= {True, True, True}
```

The standard central field Hamiltonian for the hydrogen atom is [55, 69, 292, 334, etc.]

$$H = \frac{P^2}{2m} - \frac{Z e^2}{r} \tag{5.5.b}$$

This Hamiltonian is used in many places in quantum mechanics. Thus, as a further illustration, let us note the commutation of the components of L with the constituents of the Hamiltonian (5.5.b). First, we need the simple expressions for the momentum squared and the radius.

```
In[35]:= PrimitiveForm↗ P̂² := -ħ² (∂̂x · ∂̂x + ∂̂y · ∂̂y + ∂̂z · ∂̂z);
         PrimitiveForm↗ R̂ := √(x² + y² + z²);
```

The commutation relations are easily verified.

```
In[37]:= {[L̂x, P̂²]₋, [L̂y, P̂²]₋, [L̂z, P̂²]₋} // PrimitiveForm // Expand
Out[37]= {0, 0, 0}

In[38]:= {[L̂x, R̂⁻¹]₋, [L̂y, R̂⁻¹]₋, [L̂z, R̂⁻¹]₋} // PrimitiveForm // Expand
Out[38]= {0, 0, 0}
```

Thus we can find a "complete set of commuting observables" for at least one of the components of L. Traditionally, we take this component to be L_z. Note also that L^2 commutes with the components of L.

```
In[39]:= PrimitiveForm↗ L̂² := L̂x · L̂x + L̂y · L̂y + L̂z · L̂z;
```

Just for the sake of interest, since it is not often shown, here is the expanded form of L^2.

```
In[40]:= L^2 // PrimitiveForm // Simplify
```

$$\text{Out[40]= } 2x\cdot\hat{\partial}_x - x^2\cdot\left(\hat{\partial}_y\right)^2 - x^2\cdot\left(\hat{\partial}_z\right)^2 + 2y\cdot\hat{\partial}_y - y^2\cdot\left(\hat{\partial}_x\right)^2 - y^2\cdot\left(\hat{\partial}_z\right)^2 + 2z\cdot\hat{\partial}_z - z^2\cdot\left(\hat{\partial}_x\right)^2 - z^2\cdot\left(\hat{\partial}_y\right)^2 + 2(xy)\cdot\hat{\partial}_x\cdot\hat{\partial}_y + 2(xz)\cdot\hat{\partial}_x\cdot\hat{\partial}_z + 2(yz)\cdot\hat{\partial}_y\cdot\hat{\partial}_z$$

Here are the explicit commutation relations of L^2 with the components of L.

```
In[41]:= {Commutator[L̂_x, L^2], Commutator[L̂_y, L^2], Commutator[L̂_z, L^2]} // PrimitiveForm // Expand
```

```
Out[41]= {0, 0, 0}
```

Thus we can form a complete set of commuting observables including the energy H, the total angular momentum L^2, and the z component of angular momentum L_z. The corresponding eigenfunctions are usually denoted $\psi_{nlm}(r, \theta, \phi)$.

Of course, any physicist would have known the results of the above calculations in the first place. Yet, it is reassuring to confirm that our computations are performing faithfully. It is also insightful to compare the ease and elegance of these calculations to those of other computer algebra packages for physics [107, 163, 214]

We have chosen to associate different functionalities with different environments. We have not, for instance, used one of the main environments of the previous subsection, `Act`, for expressing primitive forms. Overall, this demonstrates that we can taylor specific functions and environments to our exact requirements.

5.5.3 Raising and Lowering Operators, and Spherical Harmonics

In this subsection we introduce the spherical harmonics [30, 68, 292, 311, 334]. These functions are eigenfunctions of the angular momentum operator and also of our simple central field Hamiltonian (5.5.b). For illustration purposes, in the next subsection, we will apply the explicit operators so far developed to the spherical harmonics in order to verify that they satisfy the appropriate commutation relations.

Just as was the case with our creation and annihilation operators of §5.4.4 *Creation and Annihilation Operators*, we can introduce raising and lowering operators for angular momentum.

```
In[42]:= PrimitiveForm ↗ L̂ +1 := - (L̂ x + i L̂ y) / √2

         PrimitiveForm ↗ L̂ -1 := (L̂ x - i L̂ y) / √2

         PrimitiveForm ↗ L̂ 0 := L̂ z
```

The operators L_+, L_-, L_0 are sometimes called the spherical components of the angular momentum operator.

Technical Note: As with the inclusion of the factor of $\hbar$ into the angular momentum, some authors choose to insert a factor of $\sqrt{2}$ as well as a minus sign into their raising and lowering operators while others do not. Again, we follow Weissbluth. The advantage of this approach is that the raising and lowering operators are exactly the same as the spherical tensor components of the angular momentum — see [311, 334].

The spherical components of the angular momentum operator commute with the total angular momentum squared.

```
In[45]:= {[L̂ +1, L̂^2]_, [L̂ 0, L̂^2]_, [L̂ -1, L̂^2]_} // PrimitiveForm // Expand

Out[45]= {0, 0, 0}
```

Also, amongst the raising and lowering operators themselves, the following commutation relations hold.

```
In[46]:= ExpandAll @ PrimitiveForm @
           {[L̂ 0, L̂ +1]_ == L̂ +1, [L̂ 0, L̂ -1]_ == - L̂ -1, [L̂ +1, L̂ -1]_ == - L̂ 0}

Out[46]= {True, True, True}
```

We shall see that the quasi-spin operators of §7 *Tensor Calculus, Applications, and Quasi-Spin* also obey similar relations. This underscores the fact that it is useful to have the above machinery, since various authors use different sign conventions for the various different operations.

Technical Note: For instance, letting the initials CT denote "Cohen-Tannoudji" and WB denote "Weissbluth", then the conventions are related by $L_{+1}^{CT} = -\sqrt{2}\, L_{+1}^{WB}$ and $L_{-1}^{CT} = \sqrt{2}\, L_{-1}^{WB}$, while $L_0^{CT} = L_0^{WB}$.

Previously, we expressed the Hamiltonian in terms of the creation and annihilation operators. Similarly, we can express the total angular momentum in terms of the raising and lower operators.

```
In[47]:= L̂^2 == -2 L̂ +1 · L̂ -1 + L̂ 0 · (L̂ 0 - 1) // PrimitiveForm // ExpandAll

Out[47]= True
```

We can loosely describe the angular dependence of the atomic states by the spherical harmonics. (In a fuller picture, the eigenfunctions of an atom have to be modified by electron shielding and other effects. This can be handled by Hartree-Fock theory, or other more

advanced methods.) First, let us define a notation for the spherical harmonic functions in terms of tensors and the *Mathematica* function `SphericalHarmonicY`.

```
In[48]:= Y[θ_, ϕ_]ℓ_m_ := SphericalHarmonicY[ℓ, m, θ, ϕ]
```

Here are a couple of simple examples of spherical harmonics. These correspond to atomic states in the orbitals *p*, *d*, and *f*.

```
In[49]:= Y[θ, ϕ]1 -1
```

$$\text{Out[49]= } \frac{1}{2} e^{-i\phi} \sqrt{\frac{3}{2\pi}} \text{Sin}[\theta]$$

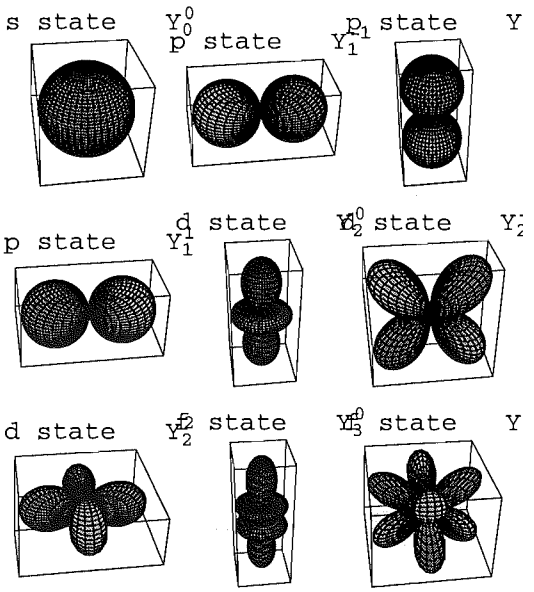
```
In[50]:= Y[θ, ϕ]2 2
```

$$\text{Out[50]= } \frac{1}{4} e^{2i\phi} \sqrt{\frac{15}{2\pi}} \text{Sin}[\theta]^2$$

```
In[51]:= Y[θ, ϕ]3 2
```

$$\text{Out[51]= } \frac{1}{4} e^{2i\phi} \sqrt{\frac{105}{2\pi}} \text{Cos}[\theta] \text{Sin}[\theta]^2$$

Actually, it is instructive to plot examples of these functions/orbitals. Although we will hide the detailed code that actually does the plotting of these functions, it is illustrative to examine the real part of the spherical harmonics, as depicted in most chemistry texts. (These can loosely be thought of as the electron wave function lobes.)



To show explicitly that the spherical harmonics commute with the components of angular momentum, it is necessary to express the spherical harmonics in terms of the Cartesian variables x, y, and z. (Or alternatively, we could recast the angular momentum operators in terms of spherical components.) There are several ways to transform the spherical harmonics,

expressed in terms of the angular variables θ and ϕ, into functions of the Cartesian variables. We choose a way which is efficient and suffices, but it is not necessarily the cleanest. The transformation between spherical and Cartesian coordinates is governed by the following equations.

$$\begin{aligned} x &= r\,\text{Sin}[\theta]\,\text{Cos}[\phi] \\ y &= r\,\text{Sin}[\theta]\,\text{Sin}[\phi] \\ z &= r\,\text{Cos}[\theta] \\ r^2 &= x^2 + y^2 + z^2 \end{aligned} \tag{5.5.c}$$

It can easily be shown from equations (5.5.c) that the following transformations are valid.

$$\begin{aligned} e^{i n \phi} &\longrightarrow \frac{(x+i\,y)^n}{(x^2+y^2)^{n/2}} \\ \text{Cos}[\theta] &\longrightarrow z/r \\ \text{Sin}[\theta] &\longrightarrow \sqrt{x^2+y^2}\,/r \\ r &\longrightarrow \sqrt{x^2+y^2+z^2} \end{aligned} \tag{5.5.d}$$

Thus, we can directly transform the "spherical" spherical harmonics into the "Cartesian" spherical harmonics by the following function.

```
In[52]:= sphericalToCartesian @ expr_ := Simplify[expr //. {
  e^Complex[0,n_Integer] ϕ :> (x + i y)^n / (x^2 + y^2)^(n/2), Cos[θ] :> z/r,
  Sin[θ] :> Sqrt[x^2 + y^2]/r, Csc[θ] :> r/Sqrt[x^2 + y^2], r :> Sqrt[x^2 + y^2 + z^2]}]
```

Our Cartesian spherical harmonics just use *Mathematica*'s built in function for spherical harmonics.

```
In[53]:= Y^R_ℓ_m_ := Y[x, y, z]^R_ℓ m

  Y[x_, y_, z_]^R_ℓ_m_ := Block[{x = x, y = y, z = z, r},
    r^ℓ Y[θ, ϕ]_ℓ m // sphericalToCartesian]
```

To demonstrate this code, here are the Cartesian spherical harmonics for the states $l = 1$, $m_l = 1$ and $l = 7$, $m_l = 5$.

```
In[55]:= Y[x, y, z]^R_1 1
```

$$\text{Out[55]=} \quad -\frac{1}{2}\sqrt{\frac{3}{2\pi}}\,(x + i\,y)$$

When the explicit Cartesian coordinates are omitted, the coordinates default to x, y, z.

```
In[56]:= Y^R_7 5
```

Out[56]= $\frac{3}{64}\sqrt{\frac{385}{2\pi}}\,(x+i\,y)^5\,(x^2+y^2-12\,z^2)$

In the following subsections we make use of these "Cartesian" spherical harmonics to demonstrate relations in the theory of angular momentum.

5.5.4 The Application of Operators

Let us demonstrate that the "Cartesian" spherical harmonics introduced in the previous subsection are eigenfunctions of L_z and L^2. To do this, we need to apply the operators L_z and L^2 to the reputed eigenfunctions. In traditional mathematics this is thought of as "function application", that is, applying an operator, viewed as a function, to an element of its domain. However, so far we have been implicitly working in just an operator algebra setting, "applying" operators to other operators via non-commutative operator multiplication.

If we want to remain in an operator algebra setting, we need some way to ensure that when appropriate, a right-most multiplication will act as "function application", that is, will act like $A \cdot B = A(B)$. There are several ways to do this. Probably the simplest method is to just "commute" (via their commutation relations) any operators through the other structures in the non-commutative product. Then, in the resulting expression, set any terms which have a remaining operator in them to zero, like so.

```
In[57]:= TerminateApplication[any_] := any /. ∂̂_ → 0
```

Using `TerminateApplication`, let us "apply" L_z to a typical spherical harmonic in order to demonstrate that it is an eigenfunction.

```
In[58]:= L̂_z · Y^R_7 5 == 5 Y^R_7 5 // PrimitiveForm // Simplify // TerminateApplication
```

```
Out[58]= True
```

Technical Note: The notion that we are basically dealing with operators is intrinsic to much of quantum mechanics. Usually the distinction between operator multiplication and operator application is overlooked since physicists will automatically "switch context". For instance $[X, P] = X.P - P.X$ is intrinsically operator based, since it is "equivalent" to $X.\partial_x - \partial_x X$. Terminating the application of the operators is equivalent to operator application by the following argument (where TA denotes `TerminateApplication`.) Since $P \cdot f(x) = P(f(x)) + f(x) \cdot P$, it follows that $\mathrm{TA}[P \cdot f(x)] = \mathrm{TA}[P(f(x))] + \mathrm{TA}[f(x) \cdot P]$. But by definition, $\mathrm{TA}[f(x) \cdot P] = 0$. Also, $\mathrm{TA}[P(f(x))] = P(f(x))$ since $P(f(x))$ is free of operators. Consequently, we can find $P(f(x))$ by computing $\mathrm{TA}[P \cdot f(x)]$. More generally for an operator Ω, it is the case that $\Omega \cdot f(x) = \Omega(f(x)) + \sum f_i(x) \cdot \Omega_i$ (for some functions f_i and some operators Ω_i), and hence equally $\Omega(f(x)) = \mathrm{TA}[\Omega \cdot f(x)]$.

But in deference to tradition, we will also present a more mathematically "standard" way and introduce a simple function, `OperatorApply`, in order to apply an element of the operator algebra to an expression.

```
In[59]:= Notation[Ω_ ? ψ_ ⟺ OperatorApply[Ω_, ψ_]]
```

Operator application is a linear operation; in fact if our operators are linear, then `OperatorApply` is multilinear.

```
In[60]:= Assign[GenericL_OpRules /. {
                  GenericConstQ → ConstQ,
                  GenericL_Op → OperatorApply,
                  GenericPlus → Plus,
                  Expand → PrimitiveForm}]
```

Here are the simple rules for operator application.

```
In[61]:= PrimitiveForm ↗ (args___ · c_?$CMultiplierQ) ⃗ ψ_ := args_nc ⃗ (c ψ)
         PrimitiveForm ↗ c_?$CMultiplierQ ⃗ ψ_ := c ψ
         PrimitiveForm ↗ args___ · ∂̂_x_ ⃗ ψ_ := args_nc ⃗ (∂_x ψ)
         PrimitiveForm ↗ ∂̂_x_ ⃗ ψ_ := ∂_x ψ
```

We can now again explicitly demonstrate that the spherical harmonics are eigenfunctions of L_z and L^2.

```
In[65]:= L̂_z ⃗ Y^R_7 5 == 5 Y^R_7 5 // PrimitiveForm // Simplify
Out[65]= True

In[66]:= L̂_z ⃗ Y^R_6 -2 == -2 Y^R_6 -2 // PrimitiveForm // Simplify
Out[66]= True

In[67]:= L̂^2 ⃗ Y^R_7 5 == 7 (7 + 1) Y^R_7 5 // PrimitiveForm // Simplify
Out[67]= True
```

Just as was the case in §5.4.4 *Creation and Annihilation Operators*, the raising and lowering operators acting on eigenkets yield new eigenkets. By various arguments, it can be shown that the following relations are obeyed by our raising and lowering operators acting on our eigenkets. In fact, these are almost the same as the relations in §2.7.4 *Example Calculations from Physics*, except that we have changed the normalization factors.

$$\begin{aligned} J_+ \cdot |j_{\hat{j}}, m_{\hat{f}_z}\rangle &= -\sqrt{\tfrac{j(j+1)-m(m+1)}{2}}\, |j_{\hat{j}}, m+1_{\hat{f}_z}\rangle \\ J_- \cdot |j_{\hat{j}}, m_{\hat{f}_z}\rangle &= +\sqrt{\tfrac{j(j+1)-m(m-1)}{2}}\, |j_{\hat{j}}, m-1_{\hat{f}_z}\rangle \\ J_0 \cdot |j_{\hat{j}}, m_{\hat{f}_z}\rangle &= +m|j_{\hat{j}}, m-1_{\hat{f}_z}\rangle \end{aligned} \tag{5.5.e}$$

To confirm that the spherical harmonics are acting as eigenfunctions, let us verify relations (5.5.e) for a few test cases. For readability let us define the following constants.

```
In[68]:= c_j_, m_, sign_ := -sign √((j (j + 1) - m (m + sign 1))/2)
```

When we act on the angular momentum eigenstate $\left|3_{\hat{J}},\, 1_{\hat{J}_z}\right\rangle$ with the raising operator, we raise the eigenstate to something proportional to $\left|3_{\hat{J}},\, 2_{\hat{J}_z}\right\rangle$.

```
In[69]:= L̂_+1 ⃗· Y^R_3 1 == c_3,1,+1 Y^R_3 2 // PrimitiveForm // Simplify

Out[69]= True
```

```
In[70]:= L̂_-1 ⃗· Y^R_6 -2 == c_6,-2,-1 Y^R_6 -3 // PrimitiveForm // Simplify

Out[70]= True
```

Actually, if we remain in an operator setting the spherical harmonics obey a similar relation.

```
In[71]:= [L̂_+1 , Y^R_3 1]_- == c_3,1,+1 Y^R_3 2 // PrimitiveForm // Simplify

Out[71]= True
```

Here are two further demonstrations of this raising and lowering action.

```
In[72]:= [L̂_-1 , Y^R_5 3]_- == c_5,3,-1 Y^R_5 2 // PrimitiveForm // Simplify

Out[72]= True
```

```
In[73]:= [L̂_+1 , Y^R_6 -2]_- == c_6,-2,+1 Y^R_6 -1 // PrimitiveForm // Simplify

Out[73]= True
```

These relations confirm that the spherical harmonics are acting like spherical tensor operators of rank (l,m) — see [68, 292, 311, 334, etc.] as well as the upcoming subsection §5.5.8 *Spherical Tensor Operators*. We could continue on with various expansions, but these relations are sufficient to show that our derivative operations are working as expected and that we can perform calculations with them.

5.5.5 Prototypical Angular Momentum

Based on the previous commutation relations, let us create a generic set of rules for angular momentum. Following this, we generalize our bras and kets so that they function correctly with tensor product states or even simultaneous eigenstates.

```
In[74]:= GenericAngularMomentumAction_Rules = {
  Generic_Env ↗ Ĵ_0 · |a___, j_Ĵ, m_Ĵz, b___⟩ :=a m |a, j_Ĵ, m_Ĵz, b⟩,
  Generic_Env ↗ Ĵ^2 · |a___, j_Ĵ, b___⟩ :=a j (j+1) |a, j_Ĵ, b⟩,
  Generic_Env ↗ Ĵ_+1 · |a___, j_Ĵ, m_Ĵz, b___⟩ :=a c_j,m,+1 |a, j_Ĵ, (m+1)_Ĵz, b⟩,
```

$$Generic_{Env} \nearrow \hat{\mathfrak{J}}_{-1} \cdot \left| a\_\_\_, j\_\hat{\mathfrak{J}}, m\_\hat{\mathcal{J}}_z, b\_\_\_ \right\rangle \overset{a}{:\!\mapsto} c_{j,m,-1} \left| a, j_{\hat{\mathfrak{J}}}, (m-1)_{\hat{\mathcal{J}}_z}, b \right\rangle,$$

$$(c_{j\_Integer,m\_Integer,1})_{Rep} :> -\sqrt{\frac{j\,(j+1) - m\,(m+1)}{2}},$$

$$(c_{j\_Integer,m\_Integer,-1})_{Rep} :> +\sqrt{\frac{j\,(j+1) - m\,(m-1)}{2}}\,\};$$

As before, let us now inherit these rules to, say, the `Act` environment.

```
In[75]:= Assign[GenericAngularMomentumAction_Rules /. {
                  Generic_Env → Act}]
```

We now act on the eigenstates of the hydrogen atom to test our relations. First, we declare that n, j, m are constants.

```
In[76]:= DeclareConstant @ {n, j, m}
```

```
In[77]:= Ĵ_-1 · Ĵ_+1 · |n_Ĥ, j_Ĵ, m_Ĵz⟩
```

```
Out[77]= Ĵ_-1 · Ĵ_1 · |n_Ĥ, j_Ĵ, m_Ĵz⟩
```

```
In[78]:= Act @ %
```

$$\text{Out[78]=}\ -\frac{1}{2}\,(j\,(1+j) - m\,(1+m))\left|n_{\hat{H}}, j_{\hat{\mathfrak{J}}}, m_{\hat{\mathcal{J}}_z}\right\rangle$$

We can confirm that J^2 is working according to the expansion shown in the previous subsection, that is, $J^2 = J_0(J_0 - 1) - 2\,J_+\,J_-$.

```
In[79]:= (-2 Ĵ_+1 · Ĵ_-1 + Ĵ_0 · (Ĵ_0 - 1)) · |n_Ĥ, j_Ĵ, m_Ĵz⟩ // Act // Simplify
```

$$\text{Out[79]=}\ j\,(1+j)\left|n_{\hat{H}}, j_{\hat{\mathfrak{J}}}, m_{\hat{\mathcal{J}}_z}\right\rangle$$

```
In[80]:= Ĵ^2 · |n_Ĥ, j_Ĵ, m_Ĵz⟩ // Act // Simplify
```

$$\text{Out[80]=}\ j\,(1+j)\left|n_{\hat{H}}, j_{\hat{\mathfrak{J}}}, m_{\hat{\mathcal{J}}_z}\right\rangle$$

Let us now add the orthonormality of the eigenkets.

```
In[81]:= Act ↗ ⟨n1_Ĥ, j1_Ĵ, m1_Ĵz| · |n2_Ĥ, j2_Ĵ, m2_Ĵz⟩ :=a δ_m1 m2 δ_j1 j2 δ_n1 n2
```

Both the inherited behavior of J and the orthonormality of the eigenstates are used to calculate matrix elements.

```
In[82]:= ⟨n_Ĥ, j_Ĵ, m_Ĵz| · Ĵ_-1 · Ĵ_+1 · |n_Ĥ, j_Ĵ, m_Ĵz⟩ // Act
```

$$\text{Out[82]=}\ \frac{1}{2}\,(-j\,(1+j) + m\,(1+m))$$

All our current machinery functions as before. For instance, if we inserted the resolution of identity between the operators in the previous calculation, we should obtain the same results. Let us verify this.

```
In[83]:= Symbolize[m'] ; Symbolize[j'] ;

In[84]:= DeclareConstant @ {m', j'};
```

$$\text{In[85]:=}\quad \hat{\mathbb{I}}_{\hat{\mathfrak{J}}} = \sum_{j',m'} \left| n_{\hat{\mathfrak{H}}},\ j'_{\hat{\mathfrak{J}}},\ m'_{\mathfrak{J}_z} \right\rangle \cdot \left\langle n_{\hat{\mathfrak{H}}},\ j'_{\hat{\mathfrak{J}}},\ m'_{\mathfrak{J}_z} \right| ;$$

$$\text{In[86]:=}\quad \left\langle n_{\hat{\mathfrak{H}}},\ j_{\hat{\mathfrak{J}}},\ m_{\mathfrak{J}_z} \right| \cdot \hat{\mathfrak{J}}_{-1} \cdot \hat{\mathbb{I}}_{\hat{\mathfrak{J}}} \cdot \hat{\mathfrak{J}}_{+1} \cdot \left| n_{\hat{\mathfrak{H}}},\ j_{\hat{\mathfrak{J}}},\ m_{\mathfrak{J}_z} \right\rangle \text{ // Expand}$$

$$\text{Out[86]=}\quad \sum_{j',m'} \left\langle n_{\hat{\mathfrak{H}}},\ j_{\hat{\mathfrak{J}}},\ m_{\mathfrak{J}_z} \right| \cdot \hat{\mathfrak{J}}_{-1} \cdot \left| n_{\hat{\mathfrak{H}}},\ j'_{\hat{\mathfrak{J}}},\ m'_{\mathfrak{J}_z} \right\rangle \cdot \left\langle n_{\hat{\mathfrak{H}}},\ j'_{\hat{\mathfrak{J}}},\ m'_{\mathfrak{J}_z} \right| \cdot \hat{\mathfrak{J}}_{1} \cdot \left| n_{\hat{\mathfrak{H}}},\ j_{\hat{\mathfrak{J}}},\ m_{\mathfrak{J}_z} \right\rangle$$

```
In[87]:= Act @ %
```

$$\text{Out[87]=}\quad \frac{1}{2}\,(-j\,(1+j)+m\,(1+m))$$

Of course, this last result is the same as that which we obtained without using the resolution of identity. In short, all of our previous machinery is applicable.

Let us create a prototypical set of rules for expressing the Cartesian components of generic angular momentum in terms of its corresponding spherical operator components.

$$\text{In[88]:=}\quad \textit{Generic}\text{AngularMomentumSpherical}_{\text{Rules}} = \Big\{ \textit{Generic}_{\text{Env}} \nearrow \left(\hat{\mathfrak{J}}_x\right)_{\text{Eqn}} :\mapsto \left(\hat{\mathfrak{J}}_{-1} - \hat{\mathfrak{J}}_{+1}\right) \Big/ \sqrt{2}\,,$$
$$\textit{Generic}_{\text{Env}} \nearrow \left(\hat{\mathfrak{J}}_y\right)_{\text{Eqn}} :\mapsto \mathrm{i}\left(\hat{\mathfrak{J}}_{-1} + \hat{\mathfrak{J}}_{+1}\right) \Big/ \sqrt{2}\,,$$
$$\textit{Generic}_{\text{Env}} \nearrow \left(\hat{\mathfrak{J}}_z\right)_{\text{Eqn}} :\mapsto \hat{\mathfrak{J}}_0 \Big\};$$

As usual, let us assign these rules.

```
In[89]:= Assign[GenericAngularMomentumSpherical_Rules /. {
              Generic_Env → Act}]
```

Similar to before, it is now a trivial matter to calculate matrix elements. For instance,

```
In[90]:= Table[⟨n_ℌ, 2_𝔍, m'_{𝔍_z}| · 𝔍̂_x · |n_ℌ, 2_𝔍, m_{𝔍_z}⟩, {m', -2, 2}, {m, -2, 2}] //
           Act // MatrixForm
```

Out[90]//MatrixForm=

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & \sqrt{\frac{3}{2}} & 0 & 0 \\ 0 & \sqrt{\frac{3}{2}} & 0 & \sqrt{\frac{3}{2}} & 0 \\ 0 & 0 & \sqrt{\frac{3}{2}} & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

5.5.6 The Addition of Angular Momenta

We next demonstrate our implementation of angular momentum by inheriting the relations of angular momentum to two different operators and then proceed to couple these operators together.

We start by introducing the prototypical normal ordering rules for a generic angular momentum. (These are based on the commutation relations shown in §5.5.3 *Raising and Lowering Operators and Spherical Harmonics.*

```
In[91]:= GenericAngularMomentumCommutation_Rules = {
           Generic_Env ↗ (Ĵ_0 · Ĵ_1)_hp :=a Ĵ_+1 · Ĵ_0 + Ĵ_+1,
           Generic_Env ↗ (Ĵ_-1 · Ĵ_0)_hp :=a Ĵ_0 · Ĵ_-1 + Ĵ_-1,
           Generic_Env ↗ (Ĵ_-1 · Ĵ_1)_hp :=a Ĵ_+1 · Ĵ_-1 + Ĵ_0};
```

Technical Note: In the above generic rule set we avoid using the index `+1`, and instead use `1` since the `HoldPattern` stops `Plus[1]` evaluating to `1`.

It is convenient to join all our prototypical rules together to yield an overall set of rules for generic angular momentum.

```
In[92]:= GenericAngularMomentum_Rules :=
           GenericAngularMomentumCommutation_Rules +{}
             GenericAngularMomentumAction_Rules +{}
             GenericAngularMomentumSpherical_Rules ;
```

Let us now create two angular momenta, namely L and S. We will add these two momenta together to form J, that is, $J = L + S$. Thus, let us create a new environment where we can resolve the components of J into its L and S constituents.

```
In[93]:= Assign[GenericAngularMomentum_Rules /. {J → L, Generic_Env → Resolve}]
         Assign[GenericAngularMomentum_Rules /. {J → S, Generic_Env → Resolve}]
```

We would also like `Act` and `NormalOrder` to recognize these angular momenta.

```
In[95]:= Assign[Values @ Resolve /. Resolve → Act]
         Assign[GenericAngularMomentumCommutation_Rules /.
```

```
        {J → L, GenericEnv → NormalOrder}]
    Assign[GenericAngularMomentumCommutationRules /.
        {J → S, GenericEnv → NormalOrder}]
```

The angular momenta L and S are independent of one another. Thus we can transform them into a normal order.

```
In[98]:= NormalOrder ↗ Ŝ_a_ · L̂_b_ :=a L̂_b · Ŝ_a

         Act ↗ Ŝ_a_ · L̂_b_ :=a L̂_b · Ŝ_a
```

```
In[100]:= Assign[Values @ NormalOrder /. NormalOrder → Resolve]
```

Furthermore, as stated above, any component of J is resolvable into its constituent components.

```
In[101]:= Resolve ↗ Ĵ_a_ := Ŝ_a + L̂_a
```

Based on the relation previously given in §5.5.3 *Raising and Lowering Operators and Spherical Harmonics*, we can express the angular momentum squared in terms of the spherical components.

```
In[102]:= AngularMomentumSquared[J_] := Ĵ_0 · Ĵ_0 - 2 Ĵ_1 · Ĵ_-1 - Ĵ_0
```

Using this, we can confirm that the total angular momentum resolves according to the following relation.

```
In[103]:= AngularMomentumSquared[J] ==
              AngularMomentumSquared[L] + AngularMomentumSquared[S] +
                  2 L̂_0 · Ŝ_0 - 2 L̂_-1 · Ŝ_+1 - 2 L̂_+1 · Ŝ_-1 // Resolve

Out[103]= True
```

Actually, the sum involving the various raising and lowering operators turns out to be just the dot product of the vectors L and S. To see this, let us force `Dot` to be a multilinear operator in the `Resolve` environment.

```
In[104]:= Unprotect @ Dot;
          Assign[GenericMLOpRules /. {
                    GenericConstQ → ConstQ,
                    GenericMLOp → Dot,
                    GenericPlus → Plus,
                    Expand → Resolve}];
          Protect @ Dot;
```

Strangely, *Mathematica* does not automatically simplify `Dot[a,0]` to `0`. There are few situations imaginable when such a transformation is not true. Thus it was necessary to unprotect `Dot` in the above, so that the zero simplification rule would always be active. Further to the multilinear rules for `Dot`, we would like the dot product of two vector operators to be the suitable sum of their component operators.

```
In[107]:= Resolve↗ L̂_.Ŝ_ :=a L̂_x · Ŝ_x + L̂_y · Ŝ_y + L̂_z · Ŝ_z
```

Using this rule, together with the previous rules for `Resolve`, allows us to transform the Cartesian dot product into spherical components.

```
In[108]:= L̂.Ŝ // Resolve

Out[108]= -(L̂_-1 · Ŝ_1) + L̂_0 · Ŝ_0 - L̂_1 · Ŝ_-1
```

We can now simplify the previous statement for the total angular momentum squared of J in terms of the constituents of L and S.

```
In[109]:= Ĵ.Ĵ == (L̂ + Ŝ).(L̂ + Ŝ) == L̂.L̂ + Ŝ.Ŝ + 2 L̂.Ŝ // Resolve

Out[109]= True
```

Many texts in quantum mechanics provide further details on adding two angular momenta together. Since this is standard knowledge, we omit such detail here. In overview, when we add two angular momentum operators together, we obtain a new angular momentum operator. The permissible quantum numbers of the new eigenstates are constrained by the quantum numbers of the two constituent eigenstates. There are two different representations, corresponding to whether we work in the coupled or uncoupled representations. The eigenkets of the uncoupled form are $|n, l, m_l, s, m_s\rangle$ while the eigenkets of the coupled form are $|n, l, s, j, m_j\rangle$.

Because of the generality of our prototypical angular momentum, both L and S act correctly on the various eigenlabels.

```
In[110]:= DeclareConstant @ {l, ml, s, ms}

In[111]:= L̂_+1 · |n_R̂, l_L̂, ml_L̂z, s_Ŝ, ms_Ŝz⟩ // Act

                                  √(l (1 + l) - ml (1 + ml)) |n_R̂, l_L̂, (1 + ml)_L̂z, s_Ŝ, ms_Ŝz⟩
Out[111]= -  ------------------------------------------------------------------------------
                                                √2

In[112]:= Ŝ^2 · Ŝ_-1 · L̂_+1 · |n_R̂, l_L̂, m_L̂z, s_Ŝ, ms_Ŝz⟩ // Act

                                  1
Out[112]= -  √(l (1 + l) - m (1 + m)) s (1 + s)
                                  2
     √(-(-1 + ms) ms + s (1 + s)) |n_R̂, l_L̂, (1 + m)_L̂z, s_Ŝ, (-1 + ms)_Ŝz⟩
```

The next subsection looks at the subject of coupling in regards to our overall paradigm.

5.5.7 Coupled versus Uncoupled Representations

The coupling of angular momenta occurs in many problems in quantum mechanics. For instance, one of the relativistic corrections to the Schrödinger equation is governed by the interaction of the orbital angular momentum and the spin angular momentum. This interaction is through a term proportional to the operator $L.S$ in the corrected Hamiltonian. It is typically known as the spin-orbit coupling term [68, 292, 334, etc.]. The question arises as to which basis it is best to solve problems in when the spin-orbit coupling term appears in our Hamiltonian. Let us examine the commutators of the spin-orbit interaction operation.

```
In[113]:= {[Ĵ.Ĵ, L̂.Ŝ]_, [L̂.L̂, L̂.Ŝ]_, [Ŝ.Ŝ, L̂.Ŝ]_} // Expand // Resolve
Out[113]= {0, 0, 0}
```

Thus the spin orbit interaction commutes with all of the squares of the angular momenta. Moreover, we note that spin does not interact with any semicommutative multipliers, or partial-derivative operators. Once this is realized, it is trivial to show that $L.S$ commutes with both $1/R$ and P^2, and hence the unperturbed Hamiltonian.

```
In[114]:= Resolve ↗ Ŝ_ν_ · term_?SCMultiplierQ :=a= term · Ŝ_ν

          Resolve ↗ Ŝ_ν_ · ∂̂_x_ :=a= ∂̂_x · Ŝ_ν

In[116]:= {[R̂^-1, L̂.Ŝ]_, [P̂^2, L̂.Ŝ]_} // PrimitiveForm // Resolve //
              PrimitiveForm // Resolve
Out[116]= {0, 0}
```

Yet, even though the spin orbit term commutes with the Z component of the total angular momentum, it does not commute with either the spin angular momentum or the orbital angular momentum.

```
In[117]:= {[Ĵ_z, L̂.Ŝ]_, [L̂_z, L̂.Ŝ]_, [Ŝ_z, L̂.Ŝ]_} // Expand // Resolve
Out[117]= {0, L̂_-1 · Ŝ_1 - L̂_1 · Ŝ_-1, -(L̂_-1 · Ŝ_1) + L̂_1 · Ŝ_-1}
```

Thus the $L.S$ operation is compatible with the complete set of commuting observables J^2, L^2, S^2, J_z but not L^2, S^2, L_z, S_z. Since as we saw before $J^2 = L^2 + S^2 + 2\,L.S$, we can evaluate matrix elements of $L.S$ in the coupled representation. First though, since we now have eigenkets and eigenbras in our calculations which can be coupled in different ways, let us encode into a prototypical set of rules a more generalized orthonormality condition for eigenkets and eigenbras.

```
In[118]:= GenericBraKetReduction_Rules = {
            Generic_Env ↗ ⟨ℓ1___, n1_ω_, r1___| · |ℓ2___, n2_ω_, r2___⟩ :=a=
```

```
    δ_{n1 n2} ⟨ℓ1, r1| · |ℓ2, r2⟩ /; Ω ∉? {0,∞} {ℓ1, r1, ℓ2, r2},
  Ket[]_{Rep} → 1,
  Bra[]_{Rep} → 1};
```

We inherit an instance of these rules to the `Act` environment.

```
In[119]:= Assign[GenericBraKetReduction_Rules /. {
            Generic_Env → Act}]
```

Here is an example of an eigenbra acting on a coupled eigenket where some reduction is possible.

```
In[120]:= ⟨a_R̂| · |1_R̂, 3_T̂⟩ // Act

Out[120]= |3_T̂⟩ δ_{a 1}
```

The situation when the bras and kets are not exactly matched can often arise. For instance, this occurs in quantum electrodynamics when dealing with electromagnetic interactions [69, 186], and also in angular momentum — for instance, see the Wigner-Eckart theorem[311, 334].

Importantly, no reduction in the bra-ket takes place when there are dependent variables. For instance, the following bra-ket is not reducible, even though there are L and S eigenlabels in both the bra and the ket.

```
In[121]:= ⟨l_L̂, ml_{L̂z}, s_Ŝ, ms_{Ŝz}| · |l_L̂, s_Ŝ, j_Ĵ, mj_{Ĵz}⟩ // Act

Out[121]= ⟨l_L̂, ml_{L̂z}, s_Ŝ, ms_{Ŝz}| · |l_L̂, s_Ŝ, j_Ĵ, mj_{Ĵz}⟩
```

With the new handling of bra-kets, let us evaluate the matrix element of $L.S$ in the coupled representation.

```
In[122]:= Symbolize[l']; Symbolize[mj']

In[123]:= ⟨l'_L̂, s_Ŝ, j'_Ĵ, mj'_{Ĵz}| · (1/2 (Ĵ^2 - L̂^2 - Ŝ^2)) · |l_L̂, s_Ŝ, j_Ĵ, mj_{Ĵz}⟩ // Act //
            Simplify

Out[123]= 1/2 (j (1 + j) - l (1 + l) - s (1 + s)) δ_{j j'} δ_{l l'} δ_{mj mj'}
```

Thus we see that $L.S$ is diagonal in the coupled basis. Finally, from the above, we know that Δm_j must be 0, but what is the restriction on the Δm_l and the Δm_s?

```
In[124]:= Symbolize[ml']; Symbolize[ms']

In[125]:= ⟨n_R̂, l'_L̂, ml'_{L̂z}, s_Ŝ, ms'_{Ŝz}| · (L̂.Ŝ) · |n_R̂, l_L̂, ml_{L̂z}, s_Ŝ, ms_{Ŝz}⟩ // Resolve //
            Act // Simplify
```

```
Out[125]= 1/2 δ_{l l'} ( √(1 + l² + ml - ml²) √(-ms (1 + ms) + s (1 + s)) δ_{ml 1+ml'} δ_{ms -1+ms'} +
                2 ml ms δ_{ml ml'} δ_{ms ms'} +
                √(1 + l² - ml (1 + ml)) √(ms - ms² + s + s²) δ_{ml -1+ml'} δ_{ms 1+ms'} )
```

Thus we can clearly see the selection rule that $\Delta m_l = -1, 0, 1$ and $\Delta m_s = -\Delta m_l$.

Actually, the above calculations are an example of an extremely important point. In the `PrimitiveForm` environment, we defined reductions for all of our angular momentum operators, such as L_x, L_y, L_z, L_+, and L_-, in terms of their primitive constituents. Yet we can happily perform calculations with these very same structures in the `Act` environment, as well as in the `Resolve` environment. Thus, even though we state rules for these objects, they are automatically "compartmentalized". Although this should be obvious from our language modifications, it is an important point that one may not be fully cognizant of, unless one's attention is specifically drawn to it.

5.5.8 Spherical Tensor Operators

Since both the coupled and uncoupled representations form orthonormal bases, we must be able to express the kets of either basis in terms of kets of the other basis. The coefficients on these basis expansions are the Clebsch-Gordon symbols [68, 292, 334, etc.]. Here is the notation for the Clebsch-Gordon symbols.

```
In[126]:= Notation[⟨ j3_, m3_ | j1_, m1_, j2_, m2_ ⟩_CG ⇔
              ClebschGordan[{j1_, m1_}, {j2_, m2_}, {j3_, m3_}]];
          Off[ClebschGordan::"phy"];
```

Technical Note: We will not perform any sophisticated abstract computations with the ClebschGordon symbols. Yet if we were to, it would probably be beneficial to introduce non-evaluating algebraic intermediaries in order to avoid the generic evaluation of these symbols.

Now given, say, an uncoupled ket, we can express this in terms of the coupled kets by inserting the resolution of identity.

```
In[128]:= Î_ĵ · | l_l̂, ml_{L̂z}, s_ŝ, ms_{Ŝz} ⟩ // Expand

Out[128]= Σ_{j',m'} | n_n̂, j'_ĵ, m'_{Ĵz} ⟩ · ⟨ n_n̂, j'_ĵ, m'_{Ĵz} | · | l_l̂, ml_{L̂z}, s_ŝ, ms_{Ŝz} ⟩
```

If we recognize this second term as a Clebsch-Gordon symbol, we can perform the expansion. For instance, we can express the uncoupled ket with values $l = 2$, $m_l = -2$ and $s = 1/2$, $m_s = -1/2$ in terms of the coupled kets as follows.

```
In[129]:= Σ_{j'=3/2}^{5/2} Σ_{m'=-j'}^{j'} | j'_ĵ, m'_{Ĵz} ⟩ · ⟨ j', m' | 2, -2, 1/2, +1/2 ⟩_CG // Act
```

Out[129]= $-\frac{2\left|\left(\frac{3}{2}\right)_{\hat{J}},\ \left(-\frac{3}{2}\right)_{\hat{J}_z}\right\rangle}{\sqrt{5}}+\frac{\left|\left(\frac{5}{2}\right)_{\hat{J}},\ \left(-\frac{3}{2}\right)_{\hat{J}_z}\right\rangle}{\sqrt{5}}$

This state is, of course, normalized since the original uncoupled state is a member of an orthonormal basis.

```
In[130]:= %† . % // Act
```

Out[130]= 1

Actually, we can couple spherical tensor operators together just like angular momenta. For instance, here is the spherical tensor operator resulting from coupling the orbital angular momentum to the spin angular momentum.

```
In[131]:= V^K_Q_ := V^K_Q = Sum[Sum[L̂_q . Ŝ_q' ⟨K, Q | 1, q, 1, q'⟩_CG, {q', -1, 1}], {q, -1, 1}] // Act
```

If L and S are coupled to 0 angular momenta, then we obtain the following expression for the coupling.

```
In[132]:= V^0_0
```

Out[132]= $\frac{\hat{L}_{-1}\cdot\hat{S}_1}{\sqrt{3}}-\frac{\hat{L}_0\cdot\hat{S}_0}{\sqrt{3}}+\frac{\hat{L}_1\cdot\hat{S}_{-1}}{\sqrt{3}}$

This can be recognized as $\frac{-1}{\sqrt{3}}\,L.S$.

```
In[133]:= % == -1/Sqrt[3] L̂.Ŝ // Resolve
```

Out[133]= True

The result of coupling two spherical tensor operators together is another spherical tensor operator, so V must obey the spherical tensor operator relations. The relations a spherical tensor operator must obey are almost identical to those stated in (5.5.e).

$$\begin{aligned}\left[J_0, V^K_Q\right] &= Q\,V^K_Q\\ \left[J_{\pm1}, V^K_Q\right] &= \mp\sqrt{\tfrac{K(K+1)-Q(Q\pm1)}{2}}\;V^K_{Q\pm1}\end{aligned}\qquad(5.5.f)$$

Let us demonstrate that relations (5.5.f) hold, by explicitly verifying a few example cases. First we consider V^0_0, which we examined above.

```
In[134]:= Comm[Ĵ_+1, V^0_0] == 0 // Expand // Resolve
```

Out[134]= True

```
In[135]:= Comm[Ĵ_-1, V^0_0] == 0 // Expand // Resolve
```

Out[135]= True

```
In[136]:= [Ĵ_0, V^0_0]_ == 0 // Expand // Resolve

Out[136]= True
```

Thus V^0_0 transforms like a scalar, and hence by the above, so does $L.S$. Let us further examine a smattering of other cases.

```
In[137]:= [Ĵ_-1, V^2_0]_ == c2,0,-1 V^2_-1 // Expand // Resolve

Out[137]= True

In[138]:= [Ĵ_+1, V^2_-2]_ == c2,-2,+1 V^2_-1 // Expand // Resolve

Out[138]= True

In[139]:= [Ĵ_0, V^2_2]_ == 2 V^2_2 // Expand // Resolve

Out[139]= True
```

Thus we see that V^K_Q is acting as a spherical tensor operator of rank (K,Q). For illustration's sake, we could create a new tensor by coupling up V^2_Q and L_q to form, say, W^K_Q.

```
In[140]:= W^K_Q_ := W^K_Q = Sum_{q1=-2}^{2} Sum_{q2=-1}^{1} V^2_q1 · L̂_q2 ⟨K, Q | 2, q1, 1, q2⟩_CG // Act
```

Here are two components of W^K_Q.

```
In[141]:= W^3_3

Out[141]= L̂_1 · L̂_1 · Ŝ_1

In[142]:= W^3_1 // Expand

Out[142]= L̂_0 · Ŝ_1/√15 + 2 L̂_1 · Ŝ_0/√15 + 2 L̂_0 · L̂_0 · Ŝ_1/√15 +
          2 L̂_1 · L̂_-1 · Ŝ_1/√15 + 4 L̂_1 · L̂_0 · Ŝ_0/√15 + L̂_1 · L̂_1 · Ŝ_-1/√15
```

The operator W^K_Q must also satisfy the relations of a spherical tensor operator, since it is constructed from the spherical tensor operators V^K_Q and L_q.

```
In[143]:= [Ĵ_+1, W^3_1]_ == c3,1,+1 W^3_2 // Expand // Resolve

Out[143]= True
```

For further details on the topics raised in this subsection, consult Weissbluth[334], Thompson[311], or Butler[45].

5.5.9 Commutation Relations and Actions on EigenKets

By this stage it should be clear how to introduce commutation relations into the workings of our system. In particular, we could easily introduce the commutation relations for the raising and lowering operators with the Cartesian variables. For instance, here are the various commutation relations.

```
In[144]:= TableForm @ PrimitiveForm[{
            {[L̂+1, x]_, [L̂0, x]_, [L̂-1, x]_},
            {[L̂+1, y]_, [L̂0, y]_, [L̂-1, y]_},
            {[L̂+1, z]_, [L̂0, z]_, [L̂-1, z]_}}]
```

Out[144]//TableForm=

$-\frac{z}{\sqrt{2}}$	$i\,y$	$-\frac{z}{\sqrt{2}}$
$-\frac{i\,z}{\sqrt{2}}$	$-i\,x$	$\frac{i\,z}{\sqrt{2}}$
$-\frac{-x-i\,y}{\sqrt{2}}$	0	$\frac{x-i\,y}{\sqrt{2}}$

Similarly, we can calculate the commutation relations of the raising and lowering operators with the momenta. For instance,

```
In[145]:= [L̂+1, ∂̂x]_ // PrimitiveForm
```

Out[145]= $-\frac{\hat{\partial}_z}{\sqrt{2}}$

Coding the above relations is performed almost exactly the same as has been previously done, so it is omitted. However, slightly more challenging is the treatment of the actions of X, Y, and Z on the angular momentum eigenkets. We would do this by expressing the Cartesian variables as spherical harmonics.

```
In[146]:= Solve[{Y^R 1 0 == Y 1 0, Y^R 1 -1 == Y 1 -1, Y^R 1 1 == Y 1 1}, {x, y, z}] //
            FullSimplify
```

Out[146]= $\left\{\left\{z \to 2\sqrt{\frac{\pi}{3}}\,\mathbf{Y}_{1\,0},\ x \to \sqrt{\frac{2\pi}{3}}\left(\mathbf{Y}_{1\,-1} - \mathbf{Y}_{1\,1}\right),\ y \to i\sqrt{\frac{2\pi}{3}}\left(\mathbf{Y}_{1\,-1} + \mathbf{Y}_{1\,1}\right)\right\}\right\}$

So we consider the action of a given Cartesian variable on an eigenket by considering the action of the corresponding combination of spherical harmonics. For instance, say we are trying to determine the action of Z on an eigenket. Let us insert the resolution of identity to obtain the following.

```
In[147]:= 𝕀_ĵ · Y_{1 0} · |n_ĥ, j_ĵ, m_Ĵz⟩ // Expand
```

$$\text{Out[147]=}\quad \sum_{j',m'} \left|n_{\hat{H}},\, j'_{\hat{J}},\, m'_{\hat{J}_z}\right\rangle \cdot \left\langle n_{\hat{H}},\, j'_{\hat{J}},\, m'_{\hat{J}_z}\right| \cdot \mathbf{Y}_{1\,0} \cdot \left|n_{\hat{H}},\, j_{\hat{J}},\, m_{\hat{J}_z}\right\rangle$$

The term $\left\langle n_{\hat{H}},\, j'_{\hat{J}},\, m'_{\hat{J}_z}\right| \cdot Y_{1\,0} \cdot \left|n_{\hat{H}},\, j_{\hat{J}},\, m_{\hat{J}_z}\right\rangle$ is equal to the Clebsch-Gordon symbol $\langle\, j',\, m' \mid 1,\, 0,\, j,\, m \,\rangle$ [68, 292, 311, etc.]. By the properties of the Clebsch-Gordon symbols, this particular symbol happens to be zero unless $j-1 \leqslant j' \leqslant j+1$ and $m' = m$. Using this knowledge, we can express the overall sum as a reduced sum over just the three possible Δj's and Δm's. (We just replace the last three terms in the non-commutative product with the appropriate Clebsch-Gordon symbol, although we could easily automate this process.)

Technical Note: Strictly, we do not need to sum over the Δm's. But by doing so, we can easily generalize the following arguments to equally handle the cases for X and Y.

```
In[148]:= Sum[Sum[|n_ĥ, j'_ĵ, m'_Ĵz⟩ · ⟨j', m' | 1, 0, j, m⟩_CG, {m', m-1, m+1}], {j', j-1, j+1}] // Act
```

$$\text{Out[148]=}\quad -\frac{\sqrt{1+2\,(-1+j)}\,\sqrt{j-m}\,\sqrt{j+m}\,\left|n_{\hat{H}},\,(-1+j)_{\hat{J}},\,m_{\hat{J}_z}\right\rangle}{\sqrt{j}\,\sqrt{-1+2\,j}\,\sqrt{1+2\,j}} - \frac{\sqrt{2}\,m\left|n_{\hat{H}},\,j_{\hat{J}},\,m_{\hat{J}_z}\right\rangle}{\sqrt{j}\,\sqrt{2+2\,j}} - \frac{(-1)^{1-2\,j+2\,m}\,\sqrt{2}\,\sqrt{1+2\,(1+j)}\,\sqrt{1+j-m}\,\sqrt{1+j+m}\,\left|n_{\hat{H}},\,(1+j)_{\hat{J}},\,m_{\hat{J}_z}\right\rangle}{\sqrt{1+2\,j}\,\sqrt{2+2\,j}\,\sqrt{3+2\,j}}$$

This can be simplified somewhat as follows.

```
In[149]:= FullSimplify[#, {j > 0 && m ∈ Reals}] & /@ %
```

$$\text{Out[149]=}\quad -\frac{\sqrt{j-m}\,\sqrt{j+m}\,\left|n_{\hat{H}},\,(-1+j)_{\hat{J}},\,m_{\hat{J}_z}\right\rangle}{\sqrt{j}\,\sqrt{1+2\,j}} - \frac{m\left|n_{\hat{H}},\,j_{\hat{J}},\,m_{\hat{J}_z}\right\rangle}{\sqrt{j}\,\sqrt{1+j}} + \frac{(-1)^{-2\,j+2\,m}\,\sqrt{1+j-m}\,\sqrt{1+j+m}\,\left|n_{\hat{H}},\,(1+j)_{\hat{J}},\,m_{\hat{J}_z}\right\rangle}{\sqrt{1+j}\,\sqrt{1+2\,j}}$$

Therefore, we can define the action of Z on a given eigenket as follows.

```
In[150]:= Act↗Ẑ · |n_ĥ, j_ĵ, m_Ĵz⟩ :=a
            2 Sqrt[π/3] (-Sqrt[j-m] Sqrt[j+m] |n_ĥ, (-1+j)_ĵ, m_Ĵz⟩ / (Sqrt[j] Sqrt[1+2 j]) - m |n_ĥ, j_ĵ, m_Ĵz⟩ / (Sqrt[j] Sqrt[1+j]) +
              (-1)^(-2 j+2 m) Sqrt[1+j-m] Sqrt[1+j+m] |n_ĥ, (1+j)_ĵ, m_Ĵz⟩ / (Sqrt[1+j] Sqrt[1+2 j]))
```

The results for the actions of X and Y are similar. Using this relation we can work out specific, or indeed general, matrix elements. For instance,

```
In[151]:= Sum[⟨n_ĥ, j'_ĵ, m'_Ĵz| · Ẑ^2 · |n_ĥ, 3_ĵ, 1_Ĵz⟩, {j', m'}] // Act // Expand
```

$$\text{Out[151]=}\quad \frac{\pi}{9} + \frac{16}{9}\sqrt{\frac{2}{7}}\,\pi - \frac{1}{3}\sqrt{\frac{5}{7}}\,\pi - \frac{\sqrt{7}\,\pi}{9} - \frac{\pi}{\sqrt{21}} - \frac{8\,\pi}{9\,\sqrt{35}}$$

Technical Note: Actually, one has to be careful when dealing with general angular momentum values versus specific momentum values. For instance, if one acts on a general eigenstate with Z, one obtains terms multiplied by $1/\sqrt{j}$. If one then substitutes say $j \to 0$, one has infinities arising within the answer. In contrast, if one operates directly on the actual eigenstate of $j = 0$ with Z, these infinities never arise since they are handled in the Clebsch-Gordon code. We could avoid the spurious infinities by introducing the opposite to the Kronecker delta, namely something like $\delta_{j\neq 0}$, which would be one for unequal values and zero for equal values. However, a much simpler way, although extremely "physicist" in nature, would be to just throw out any infinities that arise in our general calculations when we substitute specific variables into the general formulas. (Of course, the infinities which we do throw away must have arisen because of the Clebsch-Gordon coefficients, or else we will obtain incorrect answers.)

We could consider finite rotations and perform calculations associated with these objects, etc. We could investigate calculations with the Wigner-Eckart theorem, etc. Or we could perform some time independent perturbation theory involving angular momenta. For instance, we could calculate the energy changes due to say the "Stark effect" [68, 292, 311, etc.]. (One also needs to take into account degeneracy to perform this.) However, the work presented herein should provide sufficient background such that the general ideas on how to proceed are clear.

In[152]:=

5.6 Conclusions

In this chapter we have used examples in quantum mechanics to illustrate the language modifications we developed in the previous chapter. It should be evident that using our inheritance paradigm leads to an elegant, generic, and practical solution to many of the problems one faces when tackling such a subject.

Throughout this chapter we have built up several complex environments such as `Act` and `PrimitiveForm`. These environments have many environment rules attached to them, yet within each environment all the rules functioned according to their standard *Mathematica* evaluation sequence. If we had to emulate the same behavior with rewrite rules, it would be hard to ensure that some replacements did not occur before others. That is, we sometimes need to "evaluate" the internal parts of an expression before "applying" other functions. For instance, consider the case under `PrimitiveForm` when a partial derivative is acting say on a raising operator. If the raising operator has not been "fully evaluated", then erroneous calculations would be returned. With our inheritance paradigm this cannot take place, since evaluation occurs according to the standard *Mathematica* evaluation sequence. This observation was not given in the language modifications chapter since we did not then have any pertinent examples that highlighted this behavior to the degree that we do in this chapter.

From an intuitive standpoint, it should not be possible to specify the rules for something outside its environment. For our most basic operations, `Times` and `Plus`, we have under expansion the following sort of rule $a(b+c) \to a\,b + a\,c$; yet under simplification, the exact opposite transformation is true. Thus intuitively, many rules are only valid in specific environments. If our working paradigm is based on rewrite rules, we need to somehow include environments

into our rewrite rules in an intuitive way. The language modifications fulfill this goal, as well as others. They provide an elegant basis for a prototype based object-oriented system.

I believe it is insufficient just to have a general "does-not-commute" type. There are just too many structures which do not fit this categorization. For instance, Dirac gamma matrices commute with all normal spatial functions but not with other Dirac gamma matrices. Or for example, our spin angular momenta of the previous section commute with all variables like x,y, and z, as well as with partial derivatives of these variables. Yet the components of spin do not inter-commute. Thus, generally, the interrelations between structures are too complex for a general "does-not-commute" type to be useful.

In our work we have not created a complex hierarchy of commutativity and non-commutativity. However, for the hierarchies that do exist, it would be entirely possible to set them up within our inheritance paradigm. Indeed, such hierarchies are the epitome of object-oriented programming.

There are certainly some gaps in the implementations above. Abstract sums are not reindexed. Handling of time is not as elegant as one might like. Bras and kets are not commuted through each other when they contain independent eigen-labels. Operators have not automatically been set up to work on the bras when relations exist for their action on the kets. There are many other small quibbles one could raise. Yet overall, the basic structure is sound. Any of the particular gaps just mentioned can be fairly easily accommodated. Actually, in practice, abstract sum reindexing is not that common since most sums are usually tensorial and of the Einstein summation kind.

The largest caveat with using our inheritance paradigm as we have presented it is that it may be possible when performing multiple inheritances to inadvertently inherit structures in such a way that the rules are reordered to cause a conflict. Yet this appears to happen only occasionally in practice. Moreover, if one had access to the internals of *Mathematica*, one could develop rule precedences to circumvent such problems.

In some situations in symbolic computation, the expressions being manipulated are extremely large. In these situations it is often beneficial to use "tightly focused" functions, that is, functions that are designed for a single task. In other situations we would like our functions to be as general as possible. Both of these approaches are compatible with the advocated inheritance paradigm. That is, given a collection of "tightly focused" environments, we can easily inherit these environments to a new broad spectrumed environment that will subsequently contain the cumulative functionality of the "tightly focused" environments.

In summary, our paradigm of inheritance has been used extensively throughout this section, and the workings of the calculations presented fit neatly into this paradigm. The work in this section provides a base upon which calculations in quantum mechanics can be performed. Indeed, in the author's humble opinion, the style of calculations presented within this paradigm and the uniformity of the methods and structures involved make the whole presentation of quantum mechanics in this chapter extremely elegant — more elegant than that offered using any other competing system.

Chapter 6
An Algorithm for Tensor Simplification

6.1 Introduction

6.1.1 Background, History, Other Packages, and Goals

In this chapter we present in detail an algorithm for finding the canonical representation for an indicial tensor expression. It is widely known that because of the particular symmetries of tensors, any expression involving tensors can often be represented in many equivalent ways. It is a non-trivial problem to determine if two expressions involving tensors are equivalent. Thus it is necessary to have an algorithm which will determine the canonical or normal form for a expression involving tensors or tensorial expressions.

The problem is usually attacked in two different ways.

(*i*) The algorithm uses heuristics to try to find a smallest form, or

(*ii*) The algorithm uses a systematic procedure which ensures that the "smallest" representation is found.

The approach of (*i*) is intuitively not as "clean" as that of (*ii*). Certainly on the computational side of things, it is quite important to have a general algorithm for canonicalizing tensorial expressions. Our algorithm takes the second approach with some novel twists. In this section we will give an extremely brief outline of the strategy underlying the major steps. However, first the capabilities of several systems should be mentioned.

At last count, there are at least 7 other *Mathematica* packages for tensor analysis and other fields that involve indicial manipulation[56, 174, 192, 202, 249, 258, 303]. Moreover, there are other packages for other computer algebra systems such as Reduce[144, 172, 287], Macsyma[165, 218, 219, 220], and Maple[79, 80, 189, 249, 266, 267, 317, 328], and indeed stand alone systems such as those in the Sheep genealogy [7, 115, 298].

The predominant *Mathematica* tensor package is Parker & Christensen's *MathTensor* [258]. It is a capable package and has many systems and metrics built into it. Its main strategy for tackling tensor symmetries is to define specific rewrite rules to match instances that one can work out

initially by hand. Then, as rules become more complex, the rules can be built entirely within the system itself. In our rough classification above, *MathTensor* falls into category (*i*), and there are certain strengths to this approach. However, the biggest hurdle is that potentially, as expressions become large, *MathTensor* has to work very hard to try all possible matches in an effort to make reductions to an expression. For instance, to simplify tensor expressions having terms involving products of up to three Riemann tensors, *MathTensor* defines some 43 rewrite rules, some of which are quite involved. Moreover, if we need to consider products of 4 Riemann tensors, we have to add yet more rules, and so on. This process is awkward and error prone. Consequently, when the user manually transcribes new rules into the system, they must be extremely cautious since they alone must ensure the large number of rules they add are all correct. This is in contrast to the algorithm we develop where we need only specify five simple symmetries for the single Riemannian tensor factor. All other "derived" symmetries are naturally handled by the canonicalization algorithm.

Another useful package is Lee's package Ricci[202]. This also takes an approach that would fall into category (*i*) since it mainly uses heuristics. Other notable packages that fall into category (*i*) are GRTensor II[249] and those in the Sheep genealogy [7, 115, 298].

One particularly striking feature about all the papers / algorithms known to the author is that they seem to treat dummy indices as unique. Intuitively, this seems like a contradiction in terms. Yet, it appears to be the case for a large number of the algorithms. Almost all have an algorithm to relabel the indices; but none, to the author's knowledge, have used this as the starting point in an algorithm that performs a search for a canonical configuration of a tensor product. For instance, Portugal's algorithm [266], which is probably the most advanced in the literature to date, only relabels at the end of the whole canonicalization algorithm. In contrast, by using an innovative technique, we efficiently perform relabeling at each stage. This *drastically* reduces our time and space constraints.

One of the first papers that took the approach of (*ii*) was [171]. However, their algorithm was hopelessly inefficient. A practical working package that takes approach (*ii*) is that of Riegeom by Portugal[266]. In fact, his paper in part motivated the present work. Although Portugal has no relabeling code, he does, at least somewhere buried in his step 7, generate all possible equivalent configurations. And it is this that in fact instigated the author's investigations into the matter.

The following is an attempt to distill the main ideas of the algorithm into a single paragraph. We initially transform a tensor product into a "configuration". From this "configuration" we generate all equivalent "configurations" using the "symmetries" of the tensor. In doing this, we use a simplification / trick that lets us vastly reduce the number of configurations which we need to consider. From these configurations, we pick the "smallest" one as our canonical configuration. From this configuration we reconstitute the indices to obtain a "canonical" tensor product. We can graphically summarize this process in the following diagram.

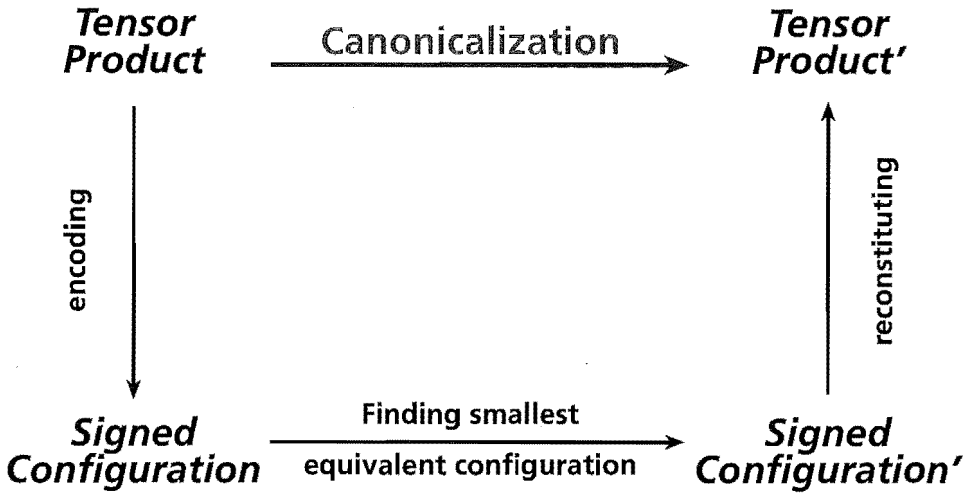


Figure 6.1.A: The canonicalization process

Although the basic form of the algorithm presented in §6.7 *The Basic Canonicalization Algorithm* is already comparatively fast, there are several key optimizations that we make to further increase its speed. For future reference, the key concepts we will introduce and develop are: a permuting and relabeling group action, the generation of configurations, the ordering on configurations, transpositional canonicalization, the canonicalization of free indices, the stabilized subgroup generators, Jointly Recursively Directional and Extrema Stabilizing generating (JRDES) sets, criteria for zero equivalence, summed indices of fixed elevations, configurations with indices of mixed class, linear symmetries, Gröbner canonicalization, and direct reduction.

It is sometimes customary in an introductory section to give a detailed overview of the subject matter to come. Such a summary has been omitted from the introduction since much of the needed terminology would be initially unknown. However, if the reader would still like to examine such a summary, then consult §6.15.1 *A Brief Summary*.

Concerning the originality of the material of this chapter and its contribution to the field, to the best of the author's knowledge, almost all of the material in the following subsections is original. An exception to this is §6.14.4 *Gröbner Bases*, which contains a brief synopsis of the theory of Gröbner bases. Also some of the background concepts in §6.8 *Generators and Group Theoretic Underpinnings* are common knowledge in group theory. It is also extremely likely that the orbit generation algorithm presented in §6.5.2 *The Algorithm for Generating Configurations* is given in the literature somewhere; however, Butler[42] does not give this algorithm in his semi-exhaustive coverage of the field. Almost certainly, the ideas, methods and theory behind the variants of the "canonicalizing algorithm" are original — in particular, transpositional canonicalization and complimentary-effect transposition pairs are original. The canonicalization of free indices is a standard idea in most algorithms for canonicalizing tensorial expressions [56, 202, 266]. Yet, the justification for our version depends upon the concept of a Jointly Recursively Directional and Extrema Stabilizing (JRDES) set. The JRDES criterion is sort of an extension of a standard concept of computational group theory, namely, that of a base with a strong generating set. However, in the case of a JRDES set, the base is sort of "flexible".

The JRDES concept is certainly unique in the context of canonicalization algorithms, and to the best of the author's knowledge is unknown / unused in present day computational group theory. For instance, neither GAP nor Magma implement such a concept. I think this is mainly because they are concerned with abstract groups and the group properties, whereas we are concerned with orbits and group actions.

After all of the ideas, concepts, theorems, proofs, examples, and code, we arrive at the final canonicalization algorithm. Its formal capabilities are that it can handle standard symmetries as well as linear symmetries and auxiliary equations. It fully incorporates special behaviors for partial derivatives. It also handles mixed class indices. Together with the notational aspects previously developed, and the forthcoming calculus aspects which make use of our language modifications, the *Tensors* package is eminently usable, understandable, and functional. Most importantly of all, comparatively, it is extremely fast.

6.2 Concepts, Notations and Background

6.2.1 Terminology and Background

Before we launch into the algorithm proper, we should first discuss the terminology used in this thesis. Tensors can be thought of as a generalization of matrices. Tensors arise in many aspects of physics, engineering, and mathematics, so it is highly desirable to have algorithms that efficiently manipulate tensor expressions. The reader of this chapter and the sequel should be familiar with the concepts surrounding tensor analysis. For a background on tensors, consult Simmonds[295], d'Inverno[81], Misner, Thorne & Wheeler[240], Ohanian[255], or any of the other standard references.

In this thesis, and in particular this chapter, we will use both the terms 'tensor' and 'tensor product', whichever is convenient, to refer to a tensor product. We will also allow a tensor product to consist of a single primitive tensor as a degenerate case. Moreover, a 'tensor expression' will in general refer to a sum of tensor products.

In this thesis we will use the term *configuration* to refer to a syntactical object consisting of a specific set and arrangement of indices of a tensor product. Sometimes such a configuration of indices will be displayed in the conventional way, that is, in relation to some tensor names. For instance, we might say $\mathbf{S}^{ijk}{}_{j}\mathbf{R}^{m}{}_{ikn}$ is a configuration of the tensor product of S and R. Similarly, $\mathbf{S}_{j}{}^{i}{}_{i}{}^{k}\mathbf{R}_{nk}{}^{jm}$ is also a configuration. Although these configurations are clearly syntactically different, they are in fact equivalent to each other when S is totally symmetric and R has the Riemann symmetries. We say that two configurations are *equivalent* if the first can be

transformed into the second via a sequence of steps using only the symmetries of the associated tensors and index relabeling.

In later stages the tensor names, such as S and R above, will be omitted from the configuration, and we will just write the configuration as the list of indices. For example, we might write the configuration $\mathbf{S}^{i\,j\,k}{}_{j}\,\mathbf{R}^{m}{}_{i\,k\,n}$ as $\{i^{\uparrow}, j^{\uparrow}, k^{\uparrow}, j^{\downarrow}, m^{\uparrow}, i^{\downarrow}, k^{\downarrow}, n^{\downarrow}\}$, where an index like $i^{\uparrow}$ indicates a high or contravariant index and $i^{\downarrow}$ indicates a low or covariant index. When we do omit the tensor names, they must of course be saved somehow so that they may be restored when we wish to display a final answer in the traditional way. For now though, the term 'configuration' will be used to refer to both a traditional 2-dimensional display of indices and to the corresponding list of indices. Actually, our algorithm will work correctly with both cartesian and non-cartesian tensors, so without loss of generality, we will sometimes give examples ignoring the high or low nature of an index.

The main purpose of the canonicalization algorithm defined in this chapter is to transform an indicial tensor expression to its canonical form. Throughout this thesis we will always assume we have a metric of some form. Although it is premature, and possibly rash, to jump straight in and use our canonicalization algorithm before we have talked about canonical forms, symmetries, generators, permutation groups, relabeling, indices or any other of the topics that will concern us in this chapter, let us proceed anyway. First let us load the package that contains the tensor notations, canonicalization algorithms, and some basic tensor symmetry descriptions.

```
In[1]:= << Tensors`
```

By default, the tensor package sets the symmetries of R to be those of the Riemann or Ricci tensor depending on the number of indices, and sets the symmetries of A and S (with two, three or four indices) to be those of a totally anti-symmetric and a totally symmetric tensor (respectively). It is now an easy matter using `Canonicalize` to show that the two syntactically different configurations of the tensor product of S and R given above do in fact represent the same tensor product.

```
In[2]:= Canonicalize[S[i,j,k,-j] R[m,-i,-k,-n]] == Canonicalize[S[j,-j,i,k] R[-n,-k,j,m]]

Out[2]= True
```

To be explicit, the relevant symmetries used, presented in conventional notation, are as follows.

$$\mathbf{R}_{i\,j\,k\,l} == -\mathbf{R}_{j\,i\,k\,l} \tag{6.2.a}$$

$$\mathbf{R}_{i\,j\,k\,l} == -\mathbf{R}_{i\,j\,l\,k} \tag{6.2.b}$$

$$\mathbf{R}_{i\,j\,k\,l} == \mathbf{R}_{k\,l\,i\,j} \tag{6.2.c}$$

$$\mathbf{S}_{i\,j\,k\,l} == \mathbf{S}_{j\,i\,k\,l} \tag{6.2.d}$$

$$\mathbf{S}_{i\,j\,k\,l} == \mathbf{S}_{i\,k\,j\,l} \tag{6.2.e}$$

$$\mathbf{S}_{i\,j\,k\,l} == \mathbf{S}_{i\,j\,l\,k} \tag{6.2.f}$$

After generating a new configuration through the use of the symmetries of the tensors, it can be the case that the new configuration is equivalent to a previous configuration by relabeling dummy indices. Let us give a simple example of this.

$$\mathbf{A}^{ij}\,\mathbf{R}_{ijkl} == -\mathbf{A}^{ij}\,\mathbf{R}_{jikl} \quad \text{(by the anti–symmetry of the Riemann tensor R (6.2.a)}$$

$$== \mathbf{A}^{ji}\,\mathbf{R}_{jikl} \quad \text{(by the anti–symmetry of A)}$$

However $\mathbf{A}^{ij}\,\mathbf{R}_{ijkl}$ is the same as the configuration $\mathbf{A}^{ji}\,\mathbf{R}_{jikl}$ via the simple interchange of the dummy index labels i and j. So we have not really found an essentially new configuration of this tensor product at all. This illustrates the problem of *dummy index relabeling*. To elaborate, in tensor simplification algorithms a large number of equivalent configurations must be generated and examined in order to find the "simplest" configuration. The larger the number of configurations generated, the longer our algorithm takes to execute. If most of the generated configurations are relabelings of one another, then we have wasted time generating a large number of redundant configurations. If we are able to represent all configurations that are equivalent to one another under relabeling by a single element, we can gain up to an $n!$ fold reduction in the number of configurations that need to be generated and considered, where n is the number of distinct dummy indices. A comparative example which strikingly illustrates this is given in §6.5.3 *The Number of Configurations*.

All tensor simplification algorithms make some attempt at trying to minimize the problem of dummy index relabeling. Our solution totally eliminates the problem.

Our goal is to create an algorithm for canonicalizing a tensor or tensor product. In later sections we will state formally what this means, but for now it suffices to give an intuitive definition. The *canonical* configuration of a tensor product T is the "smallest" configuration equivalent to T. We will formalize what is meant by the "smallest" configuration in §6.6 *The Ordering of Configurations*. In the following sections we build up the basic machinery to state and prove the algorithm. Once we can canonicalize individual terms, we can extend our overall algorithm to canonicalize sums of tensor products, respecting all of their symmetries.

6.2.2 Conventions and Initializations

Like the previous chapters, this chapter will proceed in an explanatory fashion. Since some of the underlying functions being referred to are defined in the *Tensors* package, it is convenient to use a special pedagogical version of the *Tensors* package which has no private symbols. That way there can be no confusion in this chapter about the public versus private tensor contexts. So let us quit the kernel and load the slightly modified *Tensors* package which will *only* be used in the execution of this chapter.

```
In[3]:= Quit[]
```

```
In[1]:= << TensorsPedagogical`
```

Technical Note: Actually, the only differences between `Tensors`` and `TensorsPedagogical`` is that there are no private variables in the latter package. This is accomplished by simply removing the begin private context commands and the corresponding end private context commands from the source code.

In this chapter we will make extensive use of the *Notation* package to allow the natural input and output of tensors. We will use it to define notations for almost all our basic data types, thus allowing us to describe algorithms in ways that are very close to standard mathematics. In addition, in this chapter's *Mathematica* session we will use ESC\ESC to denote set complement. Indeed, let us load up the common set of notations in order that we may freely use notations like $\equiv$, $\not\equiv$, $(\ldots)_{hp}$, $\prec_{lex}$, etc. These common notations are documented in §A.1 *Common Notations*.

```
In[2]:= << CommonNotations`
```

In general, we will denote *semantic* equality by '=' in inline text and '==' in *Mathematica* code. We uniformly denote *syntactic* equality by '≡'. Also we will usually denote a transposition $(i\,j)$ by $(i \leftrightarrow j)$, since it would be difficult and problematic to parse expressions if we used just $(i\,j)$. For instance, is $(x+y)\,(i\,f(j))$ equal to $x+y$ times a transposition or $x+y$ times i times $f(j)$? In addition, since many of the functions defined in some of the earlier sections of this chapter will be used in subsequent sections, we forgo the clearing of global variables in each section. However, as before, line numbers are still reset in each section.

Finally, let us load the *Mathematica* package `DiscreteMath`Combinatorica`` since it will be convenient to use several of the functions defined therein.

```
In[3]:= << DiscreteMath`Combinatorica`
```

The *Combinatorica* package is further documented in [299]. Usually, we will not specifically point out which functions belong to this package, rather we will just use the functions defined by this package as if they were part of standard *Mathematica*.

Up until §6.12 *Refinements for Partial Derivatives*, we will *ignore the difference* between summed (or dummy) *contravariant* and *covariant* indices. We will be using the standard syntax for permutations in *Mathematica*. For more information on permutations, cycles, and transpositions, see a standard algebra reference such as Fraleigh[114] or Biggs[21]. For algorithms used in computation group theory and permutation groups, a good reference is Butler[42]. Knuth[195, 197] also has much information on algorithm design. Skiena, who wrote the *Combinatorica* package for *Mathematica*, also published a readable reference to many typical types of algorithms arising in computer science [300].

6.3 Permutations and Configurations

In the previous section we informally discussed configurations and also gave a few examples of tensor symmetries. This section deals with formalizing the notion of configurations as well as discussing symmetries in terms of permutations. We also clarify the nature of permutations and how they will be interpreted throughout the rest of this thesis.

6.3.1 Permutations and Configurations

Symmetries on tensors are equivalent to permutations acting on configurations. For example, applying the Riemann symmetry (6.2.c) to $\mathbf{A}_{ik}\mathbf{R}_{jlmn}$ would yield $\mathbf{A}_{ik}\mathbf{R}_{mnjl}$. This gives the same effect as applying the permutation {1, 2, 5, 6, 3, 4} to the list of indices {*i*, *k*, *j*, *l*, *m*, *n*} to yield {*i*, *k*, *m*, *n*, *j*, *l*}.

```
In[1]:= Permute[{i, k, j, l, m, n}, {1, 2, 5, 6, 3, 4}]

Out[1]= {i, k, m, n, j, l}
```

Thus we can see that any symmetry applied to a tensor can be viewed as a permutation acting on a configuration. However, some of the symmetries, such as those in equations (6.2.a), (6.2.b), include an overall minus sign, so we must also include signs in our permutations and in our configurations. Thus our basic data structures are going to be signed permutations and signed configurations. Symmetries and their corresponding permutations involving a minus sign are said to act *anti-symmetrically* while those with a plus sign are said to act *symmetrically*.

As presented above, both permutations and configurations could be represented by lists of numbers. However, both conceptually and mathematically, there is a significant difference between permutations and configurations. Permutations are elements of the permutation group of the symmetries of the tensor, while configurations are the objects being acted upon by the permutations. For instance, the permutation `{1,2,4,3}` can act on `{i,j,k,i}` to yield `{i,j,i,k}`; however, there is no standard or accepted interpretation of how `{i,j,k,i}` would act on a configuration like `{a,b,c,c}`. Let us therefore make a syntactical distinction between signed configurations and signed permutations. In particular, let us maintain separate data structures for such objects and define separate notations for signed configurations as well as signed permutations.

Definition 6.3.A: A *signed configuration* is a configuration or list of indices together with an overall sign.

```
In[2]:= Notation[{indices__}+ ⟺ signedConfiguration[1, {indices__}]]
        Notation[{indices__}- ⟺ signedConfiguration[-1, {indices__}]]
```

In fact it is helpful in our explanations to also include the following more general notations.

```
In[4]:= Notation[configuration_+ ⟺ signedConfiguration[1, configuration_]]
        Notation[configuration_- ⟺ signedConfiguration[-1, configuration_]]
        Notation[configuration_sign_ ⟺ signedConfiguration[sign_, configuration_]]
```

In discussions concerning tensor products having n indices, we will let C denote the corresponding set of all possible signed configurations with n indices. Moreover, it will always be the case that when two signed configurations are equal, then they are identical; that is, $a = b \Rightarrow a \equiv b$, for any signed configurations a and b.

Definition 6.3.B: A *signed permutation* is a permutation together with a sign indicating whether the permutation's action is symmetric or anti-symmetric.

```
In[7]:= Notation[{permIndices__}+ ⟺ SignedPermutation[1, {permIndices__}]]
        Notation[{permIndices__}- ⟺ SignedPermutation[-1, {permIndices__}]]
```

As we indicated above, a symmetry on a tensor is equivalent to a signed permutation acting on a signed configuration. Permuting a signed configuration by a signed permutation will of course result in another signed configuration. Let us formally create a notation for such an action, $*_\pi$, and implement $*_\pi$.

```
In[9]:= Notation[sρ_ *π sc_ ⟺ permuteConfiguration[sρ_, sc_]]
```

The action, $*_\pi$, of the signed permutation $s\rho$ on the signed configuration sc will be to change the sign of sc according to the sign of $s\rho$ and permute the indices of the configuration part of sc according to the inverse of the permutation part of $s\rho$. (The reason why the inverse is used is explained in the next two subsections.)

```
In[10]:= SignedPermutation[signρ_, ρ_] *π configuration_signc_ :=
           Permute[configuration, inversePermutation @ ρ]signρ signc;
```

Here `inversePermutation` is a self-caching routine allowing fast inversion and defined as follows.

```
In[11]:= inversePermutation @ ρ_ := inversePermutation @ ρ =
           (Transpose @ Sort @ Transpose @ {ρ, Range @ ρ[[ten]]})[[2]]
```

The following illustrates the anti-symmetric permutation of indices 3 and 4 acting on the positive configuration `{a,b,c,d}`.

```
In[12]:= {1, 2, 4, 3}- *π {a, b, c, d}+
Out[12]= {a, b, d, c}-
```

If R is the base tensor of the signed configuration `{a, b, c, d}+`, then this result would correspond to $\mathbf{R}_{abcd} \rightarrow -\mathbf{R}_{abdc}$.

Since a large part of one of the later optimizations will make extensive use of transpositions, let us introduce at this time the notion of a signed transposition. Although signed transpositions are mathematically just a specific instance of signed permutations, we need a specialized data structure for them since we will use them so much.

Definition 6.3.C: A *signed transposition* is a transposition together with a sign indicating whether the transposition's action is symmetric or anti-symmetric.

```
In[13]:= Notation[(i_ ↔ j_)+ ⟺ SignedTransposition[1, {i_, j_}]]
         Notation[(i_ ↔ j_)- ⟺ SignedTransposition[-1, {i_, j_}]]
         Notation[(i_ ↔ j_)sign_ ⟺ SignedTransposition[sign_, {i_, j_}]]
```

The action of signed transpositions on signed configurations is just a specialization of the action of signed permutations on signed configurations. We do not actually need to define the action right now, and so delay this until §6.4.4 *Permutation and Relabeling Action*.

Finally, when it is clear from the context that we are dealing with signed configurations or signed permutations or signed transpositions, we will sometimes omit the prefix 'signed'. However, we try to stick to the convention that in *Mathematica* code a signed configuration is denoted *sc*, a signed permutation is denoted *sρ*, and a signed transposition is denoted by *sτ*.

6.3.2 Permutation Groups and Actions

Usually when one states the symmetries of a tensor, one will not state all the symmetries, but just sufficiently many from which one can generate the group of all possible symmetries of a tensor. Such a collection is called a *set of generators* for the group of symmetries. For instance, one never sees the tensor symmetry $\mathbf{R}_{abcd} \to -\mathbf{R}_{dcab}$ since it can be formed by applying symmetry (6.2.c) followed by symmetry (6.2.a).

Definition 6.3.D: Corresponding to any set of symmetry generators of a tensor, we have a corresponding set of signed permutations, typically denoted by $\mathcal{S}$. Members of $\mathcal{S}$ will be called *signed permutation generators*.

A set of signed permutation generators will of course generate a signed permutation group, as we will soon comment on. First though, we need to briefly digress into how *Mathematica* handles permutations.

Standardly one views a permutation ρ as a bijection on a finite set. That is, a permutation is a one to one and onto function of the form $\rho : \Omega \to \Omega$. Under this view, the permutation group operation is function composition. Indeed, the binary group operation we will use throughout this thesis will be function composition together with the multiplication of signs, unless otherwise specified. *Mathematica*'s `Permute` function is in accordance with function composition, that is, `Permute`$[\sigma, \rho] = \sigma \circ \rho$ for any permutations σ, ρ. More generally, we can use *Mathematica*'s `Permute` function to shuffle lists of elements (configurations). However, in

this case *Mathematica* insists we use the list as the first argument, that is, `Permute[`*list*`, σ]`. For example, consider

```
In[16]:= Permute[{a, b, c, d}, {2, 3, 4, 1}]

Out[16]= {b, c, d, a}
```

Since function composition is associative, it follows that `Permute` is associative, that is

$$\texttt{Permute[Permute[}list\texttt{, }\rho\texttt{], }\sigma\texttt{]} \equiv \texttt{Permute[}list\texttt{, Permute[}\rho\texttt{, }\sigma\texttt{]]} \tag{6.3.a}$$

As standardly known, the permutations form a group where the group operation is function composition. Since we work with signed permutations rather than permutations, we need to extend the function composition operator '$\circ$' on permutations to a corresponding binary operation on signed permutations. The extension just consists of multiplying the signs of the signed permutations to yield the new sign for the composition of the two unsigned permutations. This is formally stated in the following definition.

Definition 6.3.E: The binary operation of *signed function composition* will be denoted by '$\cdot$', and is defined by the following relations, where ρ and σ are any permutations.

$$\begin{aligned} \rho_+ \cdot \sigma_+ &= (\rho\circ\sigma)_+ \\ \rho_+ \cdot \sigma_- &= (\rho\circ\sigma)_- \\ \rho_- \cdot \sigma_+ &= (\rho\circ\sigma)_- \\ \rho_- \cdot \sigma_- &= (\rho\circ\sigma)_+ \end{aligned} \tag{6.3.b}$$

It can be easily verified that the binary operation $\cdot$ together with a set of signed permutations S generates a group of signed permutations $\langle S\rangle = \mathcal{G}$. However, including the signs in our discussions does not lead to any substantial changes in our results, so they will often be omitted even though the underlying operation is not just function composition, but rather function composition together with sign multiplication.

Technical Note: We could have used the designation $\mathcal{G}$ for the permutation group and $\mathcal{G}_\pm$ for our signed permutation group, and similarly used $C_\pm$ for our set of signed configurations. Being strictly formal, we could then identify the signed permutation group $\mathcal{G}_\pm$ with the direct product of the group $\mathcal{SI}=\langle\{1,-1\},*\rangle$ and the unsigned permutation group $\mathcal{G}$, that is, $\mathcal{G}_\pm = \mathcal{SI}\times\mathcal{G}$. However, we will use signed permutation groups and signed configurations almost exclusively throughout the rest of this thesis, so including the '$\pm$' would just lead to notational clutter.

Using the above facts, we can now show that, by design, our `permuteConfiguration` operator defined in the previous subsection, $*_\pi$, defines a group action.

Theorem 6.3.A: The operator $*_\pi : \mathcal{G}\times C \to C$ defines a *group action* of the signed permutation group $\mathcal{G}$ on the set of signed configurations C. That is,

(*i*) $e *_\pi c = c$ for all $c \in C$,

(*ii*) $\sigma *_\pi (\rho *_\pi c) = (\sigma\cdot\rho) *_\pi c$ for all $c \in C$ and all $\rho, \sigma \in \mathcal{G}$.

Proof: (*i*) Obvious from the definition of $*_\pi$. (*ii*) Up to sign, $\sigma *_\pi (\rho *_\pi c) =$ `Permute[Permute[`c, ρ^{-1}`]`, σ^{-1}`] = Permute[`c, `Permute[`ρ^{-1}, σ^{-1}`]] =` `Permute[`c, $\rho^{-1}\circ\sigma^{-1}$`] = Permute[`$c$, $(\sigma\circ\rho)^{-1}$`] =` $(\sigma\circ\rho) *_\pi c$. Thus, including signs $\sigma *_\pi (\rho *_\pi c) = (\sigma\cdot\rho) *_\pi c$. ■

6.3.3 The Nature of Permutations

Before we continue, let us clarify the nature of permutations as we will use them. Although the previous section gave the intricate details of how the signed permutations formed a group $\langle \mathcal{G}, \cdot \rangle$, and that $*_\pi$ was in fact a group action, we have not discussed the interpretation of the permutations. This is the aim of this section.

In most contexts, permutations are interpreted as affecting a *rearrangement* of a set of elements (Biggs[21], Fraleigh[114] and MacLane[216]). Usually however, as formally defined, permutations actually *change elements* rather than *change the position of elements*. Specifically, as stated before, a permutation is a one-to-one onto function $\sigma : \Omega \to \Omega$. (In fact our situation parallels that of the distinction between an active and a passive transformation.) For instance, consider the permutation denoted by `{2,3,4,1}`. It has the action of changing the elements according to 1→2, 2→3, 3→4, 4→1.

```
In[17]:= Permute[{1, 2, 3, 4}, {2, 3, 4, 1}]

Out[17]= {2, 3, 4, 1}
```

That is, the elements themselves are transformed according to

```
1  2  3  4
↓  ↓  ↓  ↓
2  3  4  1
```
(6.3.c)

However, we are more concerned with how a permutation moves elements of a set, since we will be dealing with sets made up of symbols, not numbers. As we can see from the diagram above, the permutation `{2,3,4,1}` has the effect of moving the elements at positions 1, 2, 3 and 4 to the positions 4, 1, 2 and 3 respectively. To see this effect using letters, let us permute the list `{a,b,c,d}` by `{2,3,4,1}`.

```
In[18]:= Permute[{a, b, c, d}, {2, 3, 4, 1}]

Out[18]= {b, c, d, a}
```

As is clearly evident, `b`, the element originally in position 2, has moved to position 1, that is 2→1. Likewise `c` has been moved from 3 to 2, `d` has moved from 4 to 3, and `a` has moved from 1 to 4. It is obvious that the *positions of the elements are changed* according to 2→1, 3→2, 4→3,1→4, that is, according to the rewrite rules $i^\sigma \to i$ where $\sigma =$ `{2,3,4,1}`. More generally, applying the permutation `{2,3,4,1}` to the list $\ell = \{\ell_1, \ell_2, \ell_3, \ell_4\}$, would give the list $\ell' = \{\ell_2, \ell_3, \ell_4, \ell_1\}$. In full generality, *Mathematica* permutes a list ℓ by a permutation σ to yield a list ℓ', that is `Permute`$[\ell, \sigma] = \ell'$, according to $\ell'_i = \ell_{\sigma(i)}$.

Of course, since our group action $*_\pi$ permutes by the inverse, for $\ell' = \sigma *_\pi \ell$, we have $\ell'_i =$ `Permute`$[\ell, \sigma^{-1}]_i = \ell_{\sigma^{-1}(i)}$. Thus, defining $*_\pi$ as we did results in $\ell'_{\sigma(i)} = \ell_i$, hence an index which was at position i in the configuration ℓ is moved to position $\sigma(i)$ in the resulting

configuration ℓ'. This is in accordance with viewing a permutation σ in terms of it moving an entry at position i to the position $\sigma(i)$.

In summary, we have a dichotomy. If we think of permutations as actively changing elements, then the *elements are changed* according to 1→2, 2→3, 3→4, 4→1. However, if we think of permutations as shuffling elements, then the *positions of the elements are changed* according to 2→1, 3→2, 4→3, 1→4. It is clear that the change in elements is opposite to the change in positions of the elements. Budden[39] covers in detail these contrasting interpretations of permutations. We raise this whole distinction since the core of our algorithm will use permutations in *both* of the ways described above. In particular our algorithm depends on shuffling configurations as well as relabeling configurations. These topics are both covered in the next section.

6.4 Labeling, Relabeling, and Group Actions

6.4.1 Motivation for Labeling

As raised in §6.2 *Concepts, Notations and Background,* the problem of dummy index relabeling gives rise to an $n!$ fold combinatorial explosion in the number of configurations unless steps are taken to rectify this problem. Most tensor packages do not actually treat dummy indices as true dummy indices. Typically they will include a routine to re-index a configuration, which means that they replace all the dummy indices with new dummy indices, but usually this is as far as they go.

In our algorithm we solve the dummy index problem by first labeling the indices in a systematic way, and thereafter relabeling at each stage. For the initial labeling, if a particular dummy index appears at positions i and j, then its occurrence at position i is replaced by $s_{i,j}$ and its occurrence at position j is replaced by $s_{j,i}$. For example, in $\mathbf{A}^{i\,j}\,\mathbf{R}_{i\,j\,k\,l}$ the dummy index i occurs at positions 1 and 3, so we would replace the first i by $s_{1,3}$ and the second i by $s_{3,1}$. Using this scheme, $\mathbf{A}^{i\,j}\,\mathbf{R}_{i\,j\,k\,l}$ would be labeled as $\mathbf{A}_{s_{1,3}\,s_{2,4}}\,\mathbf{R}_{s_{3,1}\,s_{4,2}\,k\,l}$. Similarly, $\mathbf{A}^{j\,i}\,\mathbf{R}_{j\,i\,k\,l}$ would also be labeled as $\mathbf{A}_{s_{1,3}\,s_{2,4}}\,\mathbf{R}_{s_{3,1}\,s_{4,2}\,k\,l}$. Thus, configurations equivalent via dummy index renaming become the same object under our labeling.

At first glance a problem appears to arise with this scheme. If we apply a permutation to a configuration as described above, we can obtain a configuration which is no longer correctly labeled. For instance, if we start with the configuration $\mathbf{A}_{s_{1,3}\,s_{2,4}}\,\mathbf{R}_{s_{3,1}\,s_{4,2}\,k\,l}$ and apply the symmetry $\mathbf{R}_{i\,j\,k\,l} \rightarrow -\mathbf{R}_{j\,i\,k\,l}$ to it, we will obtain the configuration $-\mathbf{A}_{s_{1,3}\,s_{2,4}}\,\mathbf{R}_{s_{4,2}\,s_{3,1}\,k\,l}$.

Obviously this is not correctly labeled; for example, the index $s_{3,1}$ is at position 4, not 3, and the index $s_{4,2}$ is at position 3, not 4.

We can easily remedy this problem by relabeling all the indices after we have applied a permutation. In our example, the permutation interchanged the index in position 3 with the index in position 4. Therefore, we can simply change all 3's to 4's and all 4's to 3's in the transformed configuration to return to a correctly labeled configuration.

```
In[1]:= A_{s1,3 s2,4} R_{s4,2 s3,1 k l} /. {3 → 4, 4 → 3}

Out[1]= A_{s1,4 s2,3} R_{s3,2 s4,1 k l}
```

This is now correctly labeled. It is thus extremely simple to maintain a systematically labeled configuration and still be able to apply permutations to configurations. In actuality, we will apply a permutation and relabel at the same time. Our relabeling can be accomplished by applying a small table of rewrite rules. Furthermore, these rewrite rules are only dependent on the permutation itself and are independent of the configuration being operated on. Further to this, we can even cache the relabeling rules. Therefore, relabeling is extremely fast, and in practice it takes a negligible amount of time in the algorithm. That is to say, the speed of the algorithm's execution is completely dominated by other factors.

6.4.2 Labeling

The illustration in the previous subsection motivates the strategy upon which we will embark. Let us now introduce the notations, data structures, and algorithms to accomplish this. The actual labeled configurations we will use in practice will have not only labeled summed indices but also labeled free indices as well. Therefore, let us define the data structures and notations for the summed indices, the free high (contravariant) indices, and the free low (covariant) indices as follows. (The reason why we need labeled free indices will become clear in §6.6 *The Ordering of Configurations*.)

```
In[2]:= Notation[s_{i_,j_} ⟺ s[i_, j_]]
        Notation[α_{n_}^↑ ⟺ High[α_, n_]];
        Notation[α_{n_}^↓ ⟺ Low[α_, n_]];
```

Note: unlike for free indices, until §6.12.1 *Necessity of Indices with Fixed Elevations*, we will not bother to preserve the information about whether a particular dummy index was in a high or low position. This is because normally, by using metric tensors, any tensor product having a matched pair of dummy indices in a low-high arrangement is equivalent to a tensor product with the same indices in a high-low arrangement.

Technical Note: It is not always true that we can ignore the information on whether a summed index is high or low. With expressions involving partial derivatives, we must include special handling for indices that must maintain a fixed elevation. However, we postpone this until §6.12 *Refinements for Partial Derivatives*.

The function `toSignedConfiguration` initially labels a tensor expression into the desired systematic form we will use in the rest of the algorithm. For example:

```
In[5]:= toSignedConfiguration[A^{ji} R_{jikl}]

Out[5]= {s_{1,3}, s_{2,4}, s_{3,1}, s_{4,2}, k_5^↓, l_6^↓}_+
```

Technical Note: `toSignedConfiguration` is one of several functions we introduce for pedagogical purposes only.

Definition 6.4.A: A *labeled* signed configuration is a signed configuration, where each index is either a summed index of the form $s_{i,j}$ with i being the position of the summed index and j the position of the corresponding summed index, or is of the form $f_n^{\uparrow}$ or $f_n^{\downarrow}$ with f being the free index name, n the free index position and $\uparrow$ or $\downarrow$ used to indicate the contravariant or covariant nature of the index.

Actually, we will often need to create labeled signed configurations from tensor products when discussing various topics, so it is useful to introduce a notation for `toSignedConfiguration`.

```
In[6]:= Notation[⟨tensorProduct_⟩_c ⟺ toSignedConfiguration[tensorProduct_]]
```

Here is another example of a labeled signed configuration.

```
In[7]:= signedConfig = ⟨R^{abc}_a S^{de}_{bf}⟩_c

Out[7]= {s_{1,4}, s_{2,7}, c_3^↑, s_{4,1}, d_5^↑, e_6^↑, s_{7,2}, f_8^↓}_+
```

6.4.3 Relabeling

Let us now return to the issue of relabeling after a permutation has altered a configuration. As mentioned in §6.3.3 *The Nature of Permutations*, if $\ell' = \sigma *_\pi \ell$, then $\ell'_i = \ell_{\sigma^{-1}(i)}$, or equivalently $\ell'_{\sigma(i)} = \ell_i$, that is, an index which was at position i in the configuration ℓ is moved to position $\sigma(i)$ in the resulting configuration ℓ'. However, such a move does not update any position subscript. For example, if $a_3^{\uparrow}$ is moved from position 3 to position 5, it will still be $a_3^{\uparrow}$, not $a_5^{\uparrow}$. So if the permutation σ moves the object at position i to position $\sigma(i)$, then clearly a relabeling operation should be applied in order to update the object's old location subscript from i to $\sigma(i)$. We can efficiently carry out this relabeling operation by using the rewrite rules $i \to \sigma(i)$, deleting those rules involving an i not moved by the permutation σ, that is, $i \notin Orbit(\sigma)$.

```
In[8]:= generateRelabelingRules @ σ_ :=
          DeleteCases[Table[i → σ_[[i]], {i, σ_ten}], (γ_ → γ_)_hp]
```

For instance, for the cycle `(1 2 4 3)` operating on a configuration with 8 indices, `generateRelabelingRules` just gives the rules transforming each initial position in the orbit of the permutation to its terminal position under the permutation.

```
In[9]:= generateRelabelingRules[{2, 4, 1, 3, 5, 6, 7, 8}]
```

```
Out[9]= {1 → 2, 2 → 4, 3 → 1, 4 → 3}
```

To gain speed in their use, we cache the above rules.

```
In[10]:= relabelingRules @ ρ_ :=
           relabelingRules @ ρ = generateRelabelingRules @ ρ
```

Relabeling is simply accomplished by applying the relabeling replacement rules to the permuted configuration. (We apply the replacement rules only to things at level {2} to save time and allow configurations with numbers to be manipulated.)

```
In[11]:= relabelConfiguration[sρ_SignedPermutation, configuration_sign_] :=
           Replace[configuration, relabelingRules @ sρ[[2]], {2}]sign
```

6.4.4 Permutation and Relabeling Action

We are now in the position where we can define a new group action of not only permuting the elements of a signed configuration but relabeling them as well. Let us denote this action by $*_c$.

```
In[12]:= Notation[sρ_ *c sc_ ⟺ permuteAndRelabel[sρ_, sc_]]
```

Let us define the overall action of a signed permutation on a signed configuration as simply the permutation action followed by relabeling. For later use, let us also define this action for signed transpositions.

```
In[13]:= sρ_SignedPermutation *c sc_signedConfiguration :=
           relabelConfiguration[sρ, sρ *π sc]
         sτ_SignedTransposition *c sc_signedConfiguration :=
           transposeAndRelabel[sτ, sc]
```

The function `transposeAndRelabel` will be defined shortly. Clearly for $*_c$ we have $*_c : \mathcal{G} \times C \to C$, and soon we will indeed verify that $*_c$ is a group action. This combined permuting and relabeling action is the main group action that will be used throughout this chapter. Let us illustrate this action. First we need a labeled signed configuration and a signed permutation.

```
In[15]:= signedConfig = ⟨R a b c  a S d e  b f⟩c
```

```
Out[15]= {s1,4, s2,7, c3↑, s4,1, d5↑, e6↑, s7,2, f8↓}+
```

```
In[16]:= signedPerm = {3, 4, 1, 2, 5, 6, 7, 8}+;
```

A permutation only shuffles the indices of a configuration amongst themselves.

```
In[17]:= signedPerm *π signedConfig
```

```
Out[17]= {c3↑, s4,1, s1,4, s2,7, d5↑, e6↑, s7,2, f8↓}+
```

Relabeling restores the correct labeling of the indices.

```
In[18]:= relabelConfiguration[signedPerm, %]

Out[18]= {c1↑, s2,3, s3,2, s4,7, d5↑, e6↑, s7,4, f8↓}+
```

Our action, $*_c$, simply combines permuting and relabeling.

```
In[19]:= signedPerm *c signedConfig

Out[19]= {c1↑, s2,3, s3,2, s4,7, d5↑, e6↑, s7,4, f8↓}+
```

The new configuration represents a tensor like $\mathbf{R}^{ca\ b\ de}_{\ a\ \ bf}$. (We say 'like' since using indices such as a and b is only unique up to relabeling.) Let us now formally state and prove that $*_c$ is a group action.

Theorem 6.4.A: Let $\mathcal{G}$ be a signed permutation group and let C be the set of signed configurations. Then the operator $*_c : \mathcal{G} \times C \to C$ defines a *group action* of $\mathcal{G}$ on C, that is,

(*i*) $e *_c c = c$ for all $c \in C$,

(*ii*) $\sigma *_c \rho *_c c = (\sigma \cdot \rho) *_c c$ for all $c \in C$ and all $\rho, \sigma \in \mathcal{G}$.

Proof: The only non-trivial part is (*ii*), and it is clearly sufficient to give an argument that deals with just unsigned permutations and configurations. So consider any permutations σ and ρ, and any configuration c, all of length n. Since we have already proved that $\sigma *_\pi (\rho *_\pi c) = (\sigma \cdot \rho) *_\pi c$, it only remains to focus on the relabelings of the subscripts of the summed and free indices. Up to sign, the two successive relabelings of $\rho *_\pi c$ and $\sigma *_\pi c'$, where $c' = \rho *_c c$, cause the subscripts to change according to $i \longrightarrow \rho(i) \longrightarrow \sigma(\rho(i)) = (\sigma \circ \rho)(i)$. But this gives the overall relabeling rule $i \longrightarrow (\sigma \circ \rho)(i)$ for relabeling $(\sigma \circ \rho) *_\pi c$. ■

It should be noted that the above routines for $*_\pi$ and $*_c$ are also applicable to unlabeled signed configurations. If our indices are unlabeled, then relabeling has no effect, hence $*_\pi$ and $*_c$ would have the same behavior.

Later on, we will find it necessary to define the action of a set of signed permutations (some of which might be signed transpositions) on a set of signed configurations as just the combination of the action of every possible permutation on every possible configuration. Therefore, let us overload the operator $*_c$ as follows.

```
In[20]:= S_List *c signedConfigurations_List :=
           Flatten @ Outer[permuteAndRelabel, S, signedConfigurations, 1]
```

At the theoretical level only, we will usually abbreviate extended notation such as $\mathcal{S} *_c \{a\}$ by just $\mathcal{S} *_c a$ when no confusion is possible.

Finally for completion, the special case of transposing and relabeling is heavily used in later stages, so it is desirable for the implementation to execute expeditiously. The following simple code accomplishes this goal.

```
In[21]:= transposeAndRelabel[(i_ ↔ j_)_signτ_, config_signC_] :=
           Block[{newConfig = config},
             {newConfig[[i]], newConfig[[j]]} = {config[[j]], config[[i]]};
             Replace[newConfig, {i -> j, j -> i}, {2}]_signτ signC]
```

Before we close this subsection, let us comment that there are other labeling schemes that one can implement in an attempt to gain speed. However, our labeling scheme is very simple, and our algorithm easily carries over to C or other low level languages if we need a faster implementation.

6.4.5 Equivalence Classes

In this subsection we formalize the notion of equivalence introduced in 1.2 *Concepts, Notations and Background*. It is a simple result that any group $\mathcal{H}$ together with a group action $*$ partitions the set it acts upon into equivalence classes, for instance see Fraleigh[114]. Therefore, given a set of permutations $\mathcal{S}$, the group $\langle \mathcal{S} \rangle$, acting on the set of configurations C via the group action $*_c$, partitions the configurations into equivalence classes under $\mathcal{S}$. Based on this, let us define equivalence as follows.

Definition 6.4.B: Given signed configurations $a, b \in C$ and a set of signed permutations $\mathcal{S}$, we say a is *$\mathcal{S}$–equivalent* to b if and only if $\exists\, \sigma_i \in \mathcal{S}$ such that $a = (\prod_i \sigma_i) *_c b$. We denote this equivalence by $a \sim_{\mathcal{S}} b$.

In essence we have stated that two configurations are equivalent if and only if they are in the same equivalence class generated by $\langle \mathcal{S} \rangle$. Clearly, the equivalence class of a signed configuration $a \in C$ under the set $\mathcal{S}$ is just the set of all $b \in C$ for which $b \sim_{\mathcal{S}} a$, that is, the set of configurations $\langle \mathcal{S} \rangle *_c a$.

It is clear that given two sets of permutations $\mathcal{S}_1, \mathcal{S}_2$ such that $\langle \mathcal{S}_1 \rangle = \langle \mathcal{S}_2 \rangle$, then $a \sim_{\mathcal{S}_1} b$ if and only if $a \sim_{\mathcal{S}_2} b$. This means that two configurations are equivalent independently of the particular choice of generators for a group.

Now that we have formalized the definition of equivalence for configurations, we can formally state what it means for two tensor products to be equivalent.

Definition 6.4.C: Given two tensor products with the same tensor symmetries, say T_1 and T_2, then they are *equivalent* if and only if their corresponding configurations, say t_1 and t_2, are equivalent. That is, $T_1 = T_2$ if and only if $t_1 \sim_{\mathcal{G}} t_2$, where $\mathcal{G}$ is the permutation group of the symmetries of T_1 and T_2.

Of course, for tensor expressions equivalence means semantic equality but not necessarily syntactic equality.

For a prelude of things to come, given a tensor product T having the symmetries $\mathcal{S}$, in essence our basic canonicalizing algorithm initially converts T to a signed configuration t, then

generates all configurations $a \sim_S t$, then picks the smallest one as our canonical configuration, and finally converts this back to a tensor product expression to yield our canonical representation of T.

We will use these definitions of equivalence extensively throughout the rest of this chapter. In particular, the notion of equivalence is indispensable in §6.9.1 *Transpositional Equivalence and Canonicalization* and throughout §6.9 *Transpositional Canonicalization* and §6.10 *Identically Zero Tensors.*

6.5 Generating Configurations

6.5.1 Background

Our next task is to generate all possible configurations reachable from an initial configuration by the symmetries of the corresponding tensor. Once this is complete, we can then simply choose the "smallest" configuration as our canonical configuration. Formally, given a set of generators $s_i \in S$ for a signed permutation group $\mathcal{G}$, that is $\mathcal{G} = \langle S \rangle$, and an initial signed configuration $t \in C$, we need to calculate the set $\mathcal{G} *_c t$. In group theory parlance, we must calculate the *orbit* of signed configurations equivalent to t generated by the signed permutation group $\mathcal{G}$ under the action $*_c$. Obviously, the set of all configurations reachable by the symmetries is just the set of all equivalent configurations, that is, $b \in \mathcal{G} *_c a$ if and only if $a \sim_{\mathcal{G}} b$.

This section deals with how to efficiently generate the set $\mathcal{G} *_c t$. Essentially we must use a closure algorithm, ensuring that the final set of configurations is closed under the generators. However, we must do this as efficiently as possible.

Butler[42] presents several algorithms for the generation of all elements in a permutation group $\mathcal{G}$, the most efficient of which is Dimino's algorithm. However, our goal is to calculate just $\mathcal{G} *_c t$. Of course, we could first generate all of $\mathcal{G}$ using Dimino's algorithm, and then use $\mathcal{G}$ to calculate $\mathcal{G} *_c t$. However, this would be computationally expensive, since usually $|\mathcal{G} *_c t| \ll |\mathcal{G}|$. So instead we will present a more direct method of calculating $\mathcal{G} *_c t$. Moreover, in later sections we will develop pruning techniques of our own that we can use to efficiently narrow our search space.

6.5.2 The Algorithm for Generating Configurations

The algorithm we will use proceeds as follows. Start with a signed configuration and apply the generators to this configuration to yield a new set of configurations. If we find any configurations we did not have before, then apply the generators to these newly found configurations. Repeat until we can find no more new configurations. In this way, we are iteratively forming larger and larger sets of configurations by iteratively applying the generators. This process must terminate since the final set, $\mathcal{G} *_c t$, is finite.

Let us first state the algorithm for calculating $\mathcal{G} *_c t$, namely `generateConfigurations`, and then later prove in §6.5.4 *Correctness Proof for GenerateConfigurations* that it actually does calculate all possible signed configurations reachable from t by the symmetries. The algorithm proceeds in stages: $\mathcal{F}$ is the collection of all configurations found in previous stages, $\mathcal{N}$ is the set of new configurations found at the current stage. As usual, $\mathcal{S}$ is the list of permutation generators for the group $\mathcal{G}$.

```
In[1]:= generateConfigurations[seedConfiguration_signedConfiguration, S_List] :=
          Block[{F = {}, N = {seedConfiguration}},
            While[N ≠ {},
              F = F ⋃ N;
              N = (S *c N) \ F];
            F]
```

Let us examine this algorithm in practice. Let us start with a tensor configuration from which we will generate the collection of equivalent signed configurations under the symmetries of the tensor.

```
In[2]:= seedConfig = ⟨R^abcd R^mn R_badcnm⟩c
```

```
Out[2]= {s1,8, s2,7, s3,10, s4,9, s5,12, s6,11, s7,2, s8,1, s9,4, s10,3, s11,6, s12,5}+
```

We next need a set of generators for this tensor product. We will give more details about generators in §6.7.2 *Specification of Generators*, including how to define symmetries for a tensor. In addition §6.8 *Generators and Group Theoretic Underpinnings* will cover issues to do with generators in depth. However, for now we can obtain the standard set of generators for a tensor product of defined tensors by using the function `StandardSymmetries`, which has the notation $\langle tensorProduct \rangle_{\mathcal{S}}$. Let us label this set $\mathcal{S}_{\mathcal{G}}$.

```
In[3]:= S𝒢 = ⟨R^abcd R^mn R_badcnm⟩S
```

```
Out[3]= {(1 ↔ 2)_, (3 ↔ 4)_, (5 ↔ 6)_, (7 ↔ 8)_, (9 ↔ 10)_,
          (11 ↔ 12)_, {3, 4, 1, 2, 5, 6, 7, 8, 9, 10, 11, 12}+,
          {1, 2, 3, 4, 7, 8, 5, 6, 9, 10, 11, 12}+,
          {1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 9, 10}+,
          {5, 6, 7, 8, 1, 2, 3, 4, 9, 10, 11, 12}+,
          {1, 2, 3, 4, 9, 10, 11, 12, 5, 6, 7, 8}+}
```

Let us first find the total number of equivalent configurations that can be generated and how long it takes.

```
In[4]:= generateConfigurations[seedConfig, S_G]_ten // Timing

Out[4]= {0.383333 Second, 64}
```

It would take over a page to show the 64 different equivalent configurations in the generated set, so for illustration let us show a random subset of say 5 of them. (We can use the function `RandomKSubset` of the `DiscreteMath`Combinatorica`` package to efficiently perform this task.)

```
In[5]:= RandomKSubset[generateConfigurations[seedConfig, S_G], 5]

Out[5]= {{{s1,7, s2,8, s3,12, s4,11, s5,10, s6,9, s7,1, s8,2, s9,6, s10,5, s11,4, s12,3}_,
          {s1,8, s2,7, s3,12, s4,11, s5,9, s6,10, s7,2, s8,1, s9,5, s10,6, s11,4, s12,3}_,
          {s1,6, s2,5, s3,10, s4,9, s5,2, s6,1, s7,12, s8,11, s9,4, s10,3, s11,8, s12,7}+,
          {s1,9, s2,10, s3,5, s4,6, s5,3, s6,4, s7,12, s8,11, s9,1, s10,2, s11,8, s12,7}+,
          {s1,10, s2,9, s3,5, s4,6, s5,3, s6,4, s7,11, s8,12, s9,2, s10,1, s11,7, s12,8}+}
```

For a second example, let us generate all reachable configurations from the same initial configuration but with a subset of generators small enough that we can view the entire set of results. Let us pick say the first, second and fourth generators of S_G.

```
In[6]:= generateConfigurations[seedConfig, S_G[[{1, 2, 4}]]]

Out[6]= {{{s1,7, s2,8, s3,10, s4,9, s5,12, s6,11, s7,1, s8,2, s9,4, s10,3, s11,6, s12,5}_,
          {s1,8, s2,7, s3,9, s4,10, s5,12, s6,11, s7,2, s8,1, s9,3, s10,4, s11,6, s12,5}_,
          {s1,7, s2,8, s3,9, s4,10, s5,12, s6,11, s7,1, s8,2, s9,3, s10,4, s11,6, s12,5}+,
          {s1,8, s2,7, s3,10, s4,9, s5,12, s6,11, s7,2, s8,1, s9,4, s10,3, s11,6, s12,5}+}
```

6.5.3 The Number of Configurations

As mentioned previously in §6.2 *Concepts, Notations and Background* as well as in §6.4.1 *Motivation for Labeling*, the symmetries with respect to dummy index relabeling lead to a plethora of equivalent configurations, and this can be computationally challenging to deal with unless handled properly. At this point it is instructive to compare the number of configurations generated when we do not account for dummy index relabeling with the number generated when we do; that is, to compare the number of configurations generated when we only permute at each stage with the number of configurations generated if we permute and relabel at each stage. From the previous subsection, §6.5.2, we see that when we permute and relabel, the number of configurations generated from the tensor $\mathbf{R}^{abcd}\,\mathbf{R}^{mn}{}_{ba}\,\mathbf{R}_{dcnm}$ is 64. To count the number of configurations generated when there is no relabeling, we simply start with the obvious unlabeled configuration as follows.

```
In[7]:= generateConfigurations[{a, b, c, d, m, n, b, a, d, c, n, m}_+, S_g]_ten

Out[7]= 3072
```

So we can see that without relabeling we get 3072 different configurations, which would then have to be compared to find a "smallest" configuration. Actually, it turns out that in this example the number of configurations without relabeling is the same size as the whole permutation group itself, but this is not always the case.

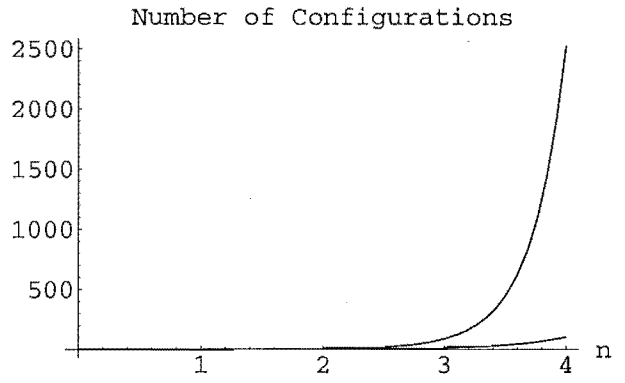
Unfortunately just searching all reachable configurations without relabeling is not sufficient to determine what the canonical configuration is since there may need to be some form of relabeling to obtain the canonical configuration. So, if one tries all possible renamings with all possible configurations, one would have to check $6! \times 3072 = 2,211,840$ configurations! However, if at each stage we relabel after permuting, then there are only 64 different configurations! This is 4 orders of magnitude fewer configurations to check. Of course there will be many duplicates in the 2, 211, 840 configurations, but there is no way to tell which will be duplicates before hand without some system like our permutation and relabeling scheme (or some other yet to be discovered routine).

Actually, the number of distinct configurations must be less than the size of our permuted and relabeled collection of 64 configurations multiplied by the number of ways to relabel the configurations, that is $n!$, where n is the number of distinct dummy indices. Certainly 3072 is less than $64 \times 6! = 46,080$. This illustrates that the number of configurations not taking into account dummy relabeling is usually less than $n!$ times the number of labeled configurations. However, the $n!$ multiplier is an upper bound on the increase from the set of equivalent labeled configurations to the larger set of all possible equivalent unlabeled configurations.

We can now very roughly describe worst case bounds on the number of configurations. Assume that every index appearing in a configuration with $2n$ indices is summed on, hence that there are n distinct summed indices. Let us also assume that the permutation generators generate S_{2n}, the full permutation group on $2n$ indices. Then the number of configurations without relabeling is $(2n)!/2^n$, that is, all permutations $(2n)!$ divided by the number of

degeneracies, which is 2^n since each distinct summed index leads to a 2 fold degeneracy. However if we relabel at each stage, then at each stage all configurations that are equivalent under dummy index renaming are represented by a single labeled configuration. Since there are $n!$ equivalent ways to relabel, the number of configurations with relabeling is bounded by $(2n)!/(n!\,2^n)$. We can easily take a look at how these functions behave.

```
In[8]:= Plot[{(2 n) !/2^n, (2 n) !/(n ! 2^n)}, {n, 0, 4},
          PlotLabel -> "Number of Configurations",
          AxesLabel -> {"n", ""}, PlotRange → All]
```




```
Out[8]= - Graphics -
```

As one can see from the above plot, and of course from inspection of $(2n)!/(n!\,2^n)$, it is obvious that in the worst case there are still many different configurations that are being generated. There are, however, additional optimizations we can and will make in the later sections to further reduce the number of superfluous configurations being generated. However, before considering these, let us return our attention to showing that the above algorithm, `GenerateConfigurations`, does in fact generate all possible configurations reachable by the given symmetries.

6.5.4 Correctness Proof for GenerateConfigurations

Let us now show that our algorithm `generateConfigurations` does in fact generate all possible configurations reachable by the symmetries of the tensor. We show that our algorithm iteratively generates a sequence of sets of configurations t, $t \cup (\mathcal{S} * t)$, ..., $\bigcup_{l=0}^{n} \mathcal{S}^l * t$, ... and thus will converge to the set of all configurations reachable by the permutation group $\mathcal{G}$, that is, $\mathcal{G} *_c t$.

Theorem 6.5.A: `generateConfigurations[`t, $\mathcal{S}$`]` generates $\mathcal{G} *_c t$ where $\mathcal{G} = \langle \mathcal{S} \rangle$.

Proof: The proof can be accomplished by a fairly simple induction argument. Let us unwind the loop in the algorithm and examine the iteratively generated sets of $\mathcal{F}$ and $\mathcal{N}$. To avoid visual clutter for the duration of this proof, we will drop the c in $*_c$ and also assume that $*$ has a higher precedence than $\backslash$, that is, $x * y \backslash z \equiv (x * y) \backslash z$. From

the algorithm the successive $\mathcal{F}$'s and $\mathcal{N}$'s are defined as follows: $\mathcal{F}_0 = \emptyset$, $\mathcal{N}_0 = \{t\}$ and for all $n \geq 1$

$$\begin{aligned} \mathcal{F}_n &= \mathcal{F}_{n-1} \cup \mathcal{N}_{n-1} \\ \mathcal{N}_n &= \mathcal{S} * \mathcal{N}_{n-1} \setminus \mathcal{F}_n \end{aligned}$$

Therefore, after the first iteration we must have,

$$\begin{aligned} \mathcal{F}_1 &= \mathcal{F}_0 \cup \mathcal{N}_0 &&= \{t\} \\ \mathcal{N}_1 &= \mathcal{S} * \mathcal{N}_0 \setminus \mathcal{F}_1 &&= \mathcal{S} * t \setminus \{t\} \end{aligned}$$

Our induction hypothesis will be:

$$\begin{aligned} \mathcal{F}_n &= \bigcup_{i=0}^{n-1} \mathcal{S}^i * t \\ \mathcal{N}_n &= \mathcal{S}^n * t \setminus \mathcal{F}_n \end{aligned}$$

Obviously the induction hypothesis is true for $n = 1$. If our induction hypothesis holds for the n^{th} case, then for the $(n+1)^{th}$ case we have

$$\begin{aligned} \mathcal{F}_{n+1} &= \mathcal{F}_n \cup \mathcal{N}_n && \text{(by definition)} \\ &= \left(\bigcup_{i=0}^{n-1} \mathcal{S}^i * t\right) \cup \left(\mathcal{S}^n * t \setminus \bigcup_{i=0}^{n-1} \mathcal{S}^i * t\right) && \text{(by induction hypothesis)} \\ &= \bigcup_{i=0}^{n} \mathcal{S}^i * t \end{aligned}$$

$$\begin{aligned} \mathcal{N}_{n+1} &= \mathcal{S} * \mathcal{N}_n \setminus \mathcal{F}_{n+1} && \text{(by definition)} \\ &= \mathcal{S} * (\mathcal{S}^n * t \setminus \mathcal{F}_n) \setminus \mathcal{F}_{n+1} && \text{(by induction hypothesis)} \\ &= \mathcal{S}^{n+1} * t \setminus \mathcal{F}_{n+1} && \text{(by argument below)} \end{aligned}$$

To justify this last step consider: for any A and B, hence $A = \mathcal{S}^n * t$ and $B = \mathcal{F}_n$, we have:

$$\begin{array}{lccccc} & \mathcal{S} * A & \supseteq & \mathcal{S} * (A \setminus B) & \supseteq & (\mathcal{S} * A) \setminus (\mathcal{S} * B) \\ \Longrightarrow & \mathcal{S}^{n+1} * t & \supseteq & \mathcal{S} * (\mathcal{S}^n * t \setminus \mathcal{F}_n) & \supseteq & \mathcal{S}^{n+1} * t \setminus \mathcal{S} * \mathcal{F}_n \\ \Longrightarrow & \mathcal{S}^{n+1} * t \setminus \mathcal{F}_{n+1} & \supseteq & \mathcal{S} * (\mathcal{S}^n * t \setminus \mathcal{F}_n) \setminus \mathcal{F}_{n+1} & \supseteq & \mathcal{S}^{n+1} * t \setminus \mathcal{S} * \mathcal{F}_n \setminus \mathcal{F}_{n+1} \end{array}$$

But $\mathcal{S} * \mathcal{F}_n \subseteq \mathcal{F}_{n+1}$, therefore the left and right bounding sets are equal, hence $\mathcal{S} * (\mathcal{S}^n * t \setminus \mathcal{F}_n) \setminus \mathcal{F}_{n+1} = \mathcal{S}^{n+1} * t \setminus \mathcal{F}_{n+1}$ as required.

Therefore, by induction $\mathcal{F}_\infty = \bigcup_i (\mathcal{S}^i * t)$; and since $\mathcal{G} = \langle \mathcal{S} \rangle$, it must be the case that $\mathcal{F}_\infty = \mathcal{G} *_c t$. Actually, since $\mathcal{G} *_c t$ is finite, there exists an n such that $\mathcal{N}_n = \{\}$, and by inspection all further iterations will then add nothing new, hence $\mathcal{F}_\infty = \mathcal{F}_n$. ■

6.5.5 Comments on Other Methods

Before we move on to considering the ordering of configurations, let us note a few points about the above algorithm. The order of the algorithm is still only $|\mathcal{G} *_c t| \times |\mathcal{S}|$. Given that the order of Dimino's algorithm is in its worst case $2\times|\mathcal{G}|$, it may be possible to form some new algorithm that has a maximum order of $2\times|\mathcal{G} *_c t|$. However, it is not immediately clear how to do this, since we are working in the set upon which the group acts, not the group itself, and some of the group properties upon which Dimino's algorithm relies on are no longer true for the sets of configurations. There may be a way, using advanced techniques in group theory or otherwise, under which one could generate all possible configurations under the group action in a more efficient way. In any case, the above suffices for our purposes, and we will find that it is fast enough in practice. Moreover, it also allows us to easily make changes. For instance, later on we will perform a transpositional canonicalization on every configuration at each stage, and this will *drastically* reduce the number of configurations generated. This transpositional canonicalization, introduced in §6.9 *Transpositional Canonicalization*, can be easily incorporated into the above closure algorithm without worrying about maintaining other group properties.

There is one particular way we could possibly make the generation of configurations more efficient, but it requires the concept of an ordering on our configurations. Roughly, it breaks up the generation of configurations into stages, and at each stage it eliminates configurations which are known to lead to non-minimal configurations. Details are given in §C.4 *Canonicalizing in Stages*. We now introduce the ordering on configurations.

6.6 The Ordering of Configurations

6.6.1 Description of the Ordering

Now that we can generate all equivalent configurations allowed by the symmetries, let us introduce an ordering on the different configurations so that we can pick the "smallest" as our canonical configuration. We would like our tensors to be as "close" as possible to the following form:

$$\mathbf{T}^{\substack{free\ summed \\ summed\ free}}$$

Here the free high indices are bunched together, followed by the summed high, then the summed low and finally the free low. If one is dealing with a tensor product, then the ideal configuration will still have the same bunching of indices, but with the names of the various tensors interspersed in some definite order, say alphabetically increasing order, with two

tensors having the same name but a different number of arguments ordered say according to the number of arguments.

Also, within each of the four groups of indices, the indices need to be in some definite order. We choose lexicographically increasing order. Others, such as Portugal[266], choose a more sophisticated ordering.

Any system of comparing two configurations will of course have to make compromises as to how heavily we weight bunching versus lexicographic ordering or some other factor. Thus, many reasonable orderings are possible, and the algorithms in this chapter can be modified to accommodate many different orderings. However, we will consider only one such ordering, the one we find most convenient for our computational purposes.

Our ordering occurs in four levels of relative importance. First it deals with the free high indices, then the free low indices, then summed indices with summed high before summed low being considered better, and finally lexicographically ordered within the summed indices.

First we focus on the free high indices. In rough terms, for our ordering, the further *left* any free high index, the better; for example, $\mathbf{T}^{a\,i}{}_{i} < \mathbf{T}^{i\,a}{}_{i}$. Our ordering also takes into account the lexicographical order of the free indices; for example, $\mathbf{S}^{a\,b} < \mathbf{S}^{b\,a}$. Next, among configurations with their free high indices in the same place, we focus on the free low indices. For our ordering, the further *right* any free low index, the better; for example $\mathbf{T}^{i}{}_{i\,b} < \mathbf{T}^{i}{}_{b\,i}$. Also, as with free high indices, our ordering takes into account the lexicographical order of the free low indices; for example, $\mathbf{S}_{a\,b} < \mathbf{S}_{b\,a}$.

Next, among configurations with the same free indices in the same places, we focus on the summed indices. For our ordering, the more summed up indices one has before any summed low, the better; for example, $\mathbf{T}^{i\,j}{}_{i\,j} < \mathbf{T}^{i\;\;j}{}_{i\;\;j}$. Finally, among configurations which are the same according to the first three criteria, we take into account the lexicographical ordering of the summed indices; for example $\mathbf{T}^{i\,j}{}_{i\,j} < \mathbf{T}^{i\,j}{}_{j\,i}$.

Obviously our whole ordering must be defined not on the tensors above but on their corresponding signed configurations. For instance, corresponding to $\mathbf{T}^{i\,j}{}_{i\,j} < \mathbf{T}^{i\;\;j}{}_{i\;\;j}$ we must have

$$\overline{\{s_{1,3},\, s_{2,4},\, s_{3,1},\, s_{4,2}\}}_{+} < \{s_{1,2},\, s_{2,1},\, s_{3,4},\, s_{4,3}\}_{+} \tag{6.6.a}$$

In our system of ordering, we ignore the sign of a configuration. Thus, two signed configurations differing only in sign will be considered to be equal relative to our ordering.

6.6.2 Definition of the Ordering

Actually, rather than define our ordering directly, it is more convenient to define it in terms of an "ordering" function, `signedConfigurationOrder`, to be denoted by $\lessgtr_c$. Like *Mathematica*'s `Order` function, `signedConfigurationOrder` returns either +1, 0 or –1, depending upon whether the first configuration is less than, equal, or greater than the second, or equivalently, whether the two configurations appear in increasing order, have the same order, or are in decreasing order. It is then an easy matter to define the usual ordering relations, as we next see.

```
In[1]:= InfixNotation[≶c, signedConfigurationOrder]
        InfixNotation[≺c, signedConfigurationLess]
        InfixNotation[≼c, signedConfigurationLessEqual]
        InfixNotation[≽c, signedConfigurationGreaterEqual]
        InfixNotation[≻c, signedConfigurationGreater]
        InfixNotation[==c, signedConfigurationEquivalent]
```

```
In[7]:= a_ ≺c b_ := (a ≶c b) == 1
        a_ ≼c b_ := (a ≶c b) ≥ 0
        a_ ≽c b_ := (a ≶c b) ≤ 0
        a_ ≻c b_ := (a ≶c b) == -1
        a_ ==c b_ := (a ≶c b) == 0
```

It is convenient to have our orderings work with lists of arguments like $a \prec_c b \prec_c c$. The simple yet somewhat cryptic implementation of the extended orderings is contained in the code section.

The above definitions for $\prec_c$, $\preccurlyeq_c$, $\succcurlyeq_c$ and $\succ_c$ are natural for any reasonable version of the function `signedConfigurationOrder`. By design our chosen version will be

```
In[12]:= config1_sign1_ ≶c config2_sign2_ :=
           Block[{ordering},
             ordering = Order[Sort @ Cases[config1, _High],
                   Sort @ Cases[config2, _High]];
             If[ordering == 0, ordering = Order[Reverse @ Sort @ Cases[config2, _Low],
                   Reverse @ Sort @ Cases[config1, _Low]];
               If[ordering == 0, ordering = Order[ config1 /. s → elevation,
                     config2 /. s → elevation];
                 If[ordering == 0, ordering = Order[ config1, config2]]]];
             ordering]
```

The elevation of each $s_{i\,j}$ index is given by

```
In[13]:= elevation[i_Integer, j_Integer] := If[i < j, High, Low]
```

It is apparent from the definition just given that our ordering does not distinguish between signed configurations differing only in their signs, that is, when $config_- =_c config_+$. Consequently, $a =_c b$, does *not* necessarily imply that $a = b$. Hence, we must be cognizant that $a \preccurlyeq_c b$ strictly means that $a \prec_c b$ or $a =_c b$. The ramifications of this design decision will be further taken up in §6.6.4 *Almost Total Orderings* and §6.6.5 *Minimum Configurations*.

6.6.3 Examples of the Ordering

Let us illustrate our chosen ordering through examples. First, if possible, an ordering is determined from the free high indices. The further to the left a free high index is the better, where lexically smaller free indices take precedence. For example,

```
In[14]:= ⟨T^{a i b}_{i}⟩_c <_c ⟨T^{i a b}_{i}⟩_c

Out[14]= True
```

This is because, using $<_{lex}$ to denote lexicographic ordering, we have

```
In[15]:= {High[a, 1], High[b, 3]} <lex {High[a, 2], High[b, 3]}

Out[15]= True
```

However, if the free high indices are the same and in the same places or there are no free high indices, then we try to determine an ordering from the free low indices. For example,

```
In[16]:= ⟨T^{a}_{b c d}⟩_c <_c ⟨T^{a}_{d c b}⟩_c

Out[16]= True
```

If we still cannot order the two configurations based on the free high or free low indices, then we proceed onto ordering the configurations based upon the summed indices.

The code for the summed indices might at first seem a little strange, so we will elaborate. According to the definition, to compare two configurations with respect to the summed indices, one replaces each index of the form $s_{i,j}$ by `High` or `Low` depending upon whether $i < j$ or $i > j$, and then lexicographically compares the resulting "configurations". For example, this gives

```
In[17]:= ⟨X^{a i j}_{i j}⟩_c <_c ⟨X^{a i  j}_{i  j}⟩_c

Out[17]= True
```

This is because, using `H` = `High`, `L` = `Low`, we have

```
In[18]:= {H[a, 1], H, H, L, L} <lex {H[a, 1], H, L, H, L}

Out[18]= True
```

There are, in fact, several ways we could have implemented ordering on summed indices. However, the above code very efficiently ensures that an index like $s_{6,9}$ is considered to be smaller than $s_{6,5}$ since $s_{6,9}$ is the first index of a summed pair, whereas $s_{6,5}$ is obviously the second index of a summed pair.

To further illustrate our ordering, consider the following more complicated examples.

```
In[19]:= ⟨R^mpat R^cn_mp⟩_c ≺_c ⟨R^mpat R^cn_pm⟩_c

Out[19]= True

In[20]:= {⟨T^ia_i⟩_c ≺_c ⟨T^ai_i⟩_c, ⟨T^ia_i⟩_c ≻_c ⟨T^ai_i⟩_c}

Out[20]= {False, True}

In[21]:= ⟨T^abi_i⟩_c ≺_c ⟨T^aib_i⟩_c ≺_c ⟨T^bai_i⟩_c ≺_c ⟨T^iab_i⟩_c ≺_c ⟨T^bia_i⟩_c

Out[21]= True

In[22]:= ⟨S^abi j_i j⟩_c ≺_c ⟨S^aibj_ij⟩_c ≺_c ⟨S^aib j_i j⟩_c ≺_c ⟨S^iabj_ij⟩_c ≺_c ⟨S^iab j_i j⟩_c

Out[22]= True
```

6.6.4 Almost Total Orderings

As noted in §6.6.2 *The Definition of the Ordering*, our ordering, $\prec_c$, does not distinguish between signed configurations which differ only in their signs. Formally, our ordering, $\prec_c$, is *degenerate* in that two configurations differing only in their sign are considered the same by our ordering. In some cases, this can cause some rather peculiar effects. Consequently, definitions using the ordering $\prec_c$ have to be stated carefully. This subsection will address and clarify the issues surrounding this degeneracy.

First though, an obvious question arises. In practice, do we ever encounter two signed configurations that are exactly the same except for the sign? To answer this let us quote a theorem from §6.10.1 *Zero Equivalence in the Basic Algorithm*. For a tensor T whose signed configuration is t and whose symmetry group is $\mathcal{G}$, the following theorem is true.

Theorem 6.10.A: A tensor product T is identically zero if and only if $\exists\, a \in C$ such that $a \in \mathcal{G} *_c t$ and $-a \in \mathcal{G} *_c t$.

Thus, signed configurations which are degenerate with respect to $\prec_c$ are only encountered if the tensor under consideration is identically zero — equivalent to zero by the symmetries alone, see §6.10 *Identically Zero Tensors*. Moreover, the tensors that are identically zero are handled by our algorithm as a special separate case. Thus, almost all of the time, our ordering on signed configurations, $\prec_c$, will be a total ordering.

Definition 6.6.A: A *total ordering* on a set $\mathcal{A}$ is a relation, $\prec$, which obeys the following conditions.

(*i*) $a \prec b$ or $a = b$ or $a \succ b$, for all $a, b \in \mathcal{A}$ (trichotomy)

(*ii*) $a \prec b \prec c \Rightarrow a \prec c$, for all $a, b, c \in \mathcal{A}$ (transitivity)

However, for our relation $\prec_c$, the trichotomy condition does not hold since we cannot guarantee equality. Specifically, $a =_c b$ does not imply $a = b$. However, the properties of $\prec_c$ are sufficient for our purposes, in that $\prec_c$ is an *almost total ordering*.

Theorem 6.6.A: The ordering $\prec_c$ is an *almost total ordering* on the set of all signed configurations C, that is,

(*i*) $a \prec_c b$ or $a =_c b$ or $a \succ_c b$, for all $a, b \in C$ (almost-trichotomy)

(*ii*) $a \prec_c b \prec_c c \Rightarrow a \prec_c c$, for all $a, b, c \in C$ (transitivity)

Proof: Our definition for comparing two signed configurations is clearly equivalent to

```
c1_sign1_ ≲c c2_sign2_ :=
    Block[{ordering},
        ordering = Test1[c1, c2];
        If[ordering == 0, ordering = Test2[c1, c2]];
        If[ordering == 0, ordering = Test3[c1, c2]];
        If[ordering == 0, ordering = Test4[c1, c2]];
        ordering]
```

(6.6.b)

(*i*) Since each $\text{Test}_i[a, b]$ yields either +1, 0 or −1, obviously the answer returned must be +1, 0 or −1. That is to say, our comparison yields either $a \prec_c b$ (*ordering* = +1) or $a =_c b$ (*ordering* = 0) or $a \succ_c b$ (*ordering* = -1).

(*ii*) One can easily check for $k = 1, 2, 3$ and 4 that

❶ if $\text{Test}_k[a, b] = 0$ and $\text{Test}_k[b, c] = 1$, then $\text{Test}_k[a, c] = 1$;

❷ if $\text{Test}_k[a, b] = 1$ and $\text{Test}_k[b, c] = 0$, then $\text{Test}_k[a, c] = 1$;

❸ if $\text{Test}_k[a, b] = 1$ and $\text{Test}_k[b, c] = 1$, then $\text{Test}_k[a, c] = 1$.

Say $a \prec b$ by Test_i, that is, $\text{Test}_k[a, b] = 0$ for $k < i$ but $\text{Test}_i[a, b] = 1$. Similarly say $b \prec c$ by Test_j. Then one easily checks that if $i > j$, then case ❶ holds with $k = j$; and if $i < j$, then case ❷ holds with $k = i$; and if $i = j$, then case ❸ holds with $k = i = j$. ■

As an aside, our ordering $\prec_c$ is in fact an instance of what is sometimes called a linear quasi-ordering — see for example Becker & Weispfenning[19]. In their terminology, a quasi-order is any binary relation which is both reflexive and transitive. A linear quasi-order is a quasi-order for which all elements are comparable. It is instructive to relate the example that Becker & Weispfenning use to typify a quasi-ordering. Given $x_0 \in \mathbb{R}[0, 1]$, then $\forall f, g \in \mathbb{R}[0, 1] \to \mathbb{R}[0, 1]$ they define $f \preccurlyeq_q g$ if and only if $f(x_0) \leq g(x_0)$. It is easy to see that there exists f', g' such that $f' \preccurlyeq_q g'$ and $f' \succcurlyeq_q g'$, hence $f' =_q g'$ but $f' \neq g'$. The ordering $\preccurlyeq_q$ is an example of a linear quasi-ordering that is not a total ordering.

6.6.5 Minimum Configurations

The motivation for our concept of an 'almost total ordering' should now be apparent. Our ordering, $\prec_c$, is extremely close to a total ordering in that all signed configurations can be compared and the comparisons are consistent with each other. Also, each set of signed configurations equivalent by the symmetries of the tensor is of course finite, thus we can proceed to define a "minimum" configuration, that is, a canonical configuration. In actuality the minimum may not be unique; specifically, both $config_+$ and $config_-$ might be "minimum" configurations. Mathematically it may be dubious to speak of a minimum element if there are two minimal elements; however, we can largely ignore this difference, as we will see below.

Definition 6.6.B: Let $\mathcal{A}$ be an arbitrary non-empty set of signed configurations.

(*i*) A *minimum signed configuration* of $\mathcal{A}$ is any $a \in \mathcal{A}$ which is as small or smaller than all of the other signed configurations $b \in \mathcal{A}$: Formally, $a \preccurlyeq_c b, \forall\, b \in \mathcal{A}$.

(*ii*) $mins_c(\mathcal{A})$ is the *set of minima* of $\mathcal{A}$, that is, $\{a \in \mathcal{A} \mid \forall\, b \in \mathcal{A},\ a \preccurlyeq_c b\}$.

(*iii*) min_c is a non-unique theoretical function that picks out a (or the) minimum from each non-empty set of configurations, that is, $min_c(\mathcal{A}) \in mins_c(\mathcal{A})$.

As has been noted, the set of minima almost always consists of a single configuration. Therefore, with impunity, we will use the expression *the* minimum, where it is understood that the exceptions to this designation will already have been handled or can be handled by a simple extension.

In fact, our implementation is implicitly biased towards this interpretation in that it only ever returns a single minimum signed configuration. Let us now give our implementation of the function min_c.

```
In[23]:= Symbolize[Minc, SymbolizeRootName → "minimumConfiguration"]

In[24]:= Minc @ signedConfigurations_List :=
           Module[{smallest = signedConfigurations[[1]]},
             Scan[If[sc ≺c smallest, smallest = sc] &sc, signedConfigurations]; smallest]
```

Let us briefly comment on the underlying mathematical formalism in order to quash any question about well-foundedness. In actuality, our definitions involving almost total orderings or almost well orderings can all be handled by realizing we have a well ordering on the *equivalence classes* of the signed configurations, where two signed configurations are in the same equivalent class if their unsigned configurational part is the same. We have purposely side stepped this formality since we will extensively use totally different equivalence classes later, and any algorithm involving two entirely separate notations of equivalence would need to be detailed to the point of pedantry.

The only other reasonable alternative, in terms of formalism, would be to extend the definition of our ordering so that it distinguished between the same configurations with different signs. However, this approach would be very artificial in that our implementation would not respect

these definitions, and to extend the implementation would be both inefficient and more involved.

It should now be apparent why we have chosen not to extend our ordering. At the expense of a non-standard and somewhat eccentric definition, we have managed to semi-formalize the concepts underlying our implementation. In reality, our code never distinguishes between different signs since it would be inefficient to do so. Indeed, it will become apparent in §6.9 *Transpositional Canonicalization* that this disregard for the sign in our ordering greatly simplifies the implementation of some parts of the algorithm to come.

In the next section, §6.7 *The Basic Canonicalization Algorithm*, we present our basic version of the canonicalization algorithm, where the minimum signed configuration of the equivalent signed configurations will be the canonical signed configuration.

6.7 The Basic Canonicalization Algorithm

6.7.1 The Basic Canonicalization Algorithm

We have now presented the three cornerstones of our algorithm for canonicalizing tensor expressions, namely, our permutation and relabeling, the generation of configurations and the ordering of configurations. In this section we will put these concepts together and present a basic version of our canonicalization algorithm. As outlined previously, our algorithm will consist of the following steps.

1. Expand all sums and products in the tensor expression.

For each tensor or tensor product in the sum, perform steps 2 through 7

2. Encode the tensor product into a signed configuration.
3. Calculate the set of generators S corresponding to the symmetries of the tensor product.
4. Generate the set of all equivalent signed configurations under the generators.
5. Check to see if the tensor product is identically zero by examining all generated signed configurations. If so, return zero.
6. If the tensor product is non–zero, pick the smallest signed configuration using the ordering on configurations.
7. Reconstitute the canonical labeled signed configuration into the canonical tensor product.

Of course, there are issues yet to be tackled which have to be elucidated in later sections. However, we are at the stage where we can present the code which accomplishes exactly the

steps above. After we present the code, we will discuss the steps involved and indicate what remains to be covered.

The code that accomplishes step 1 simply maps our `BasicCanonicalize` function over sums and factors out numbers from products.

```
In[1]:= BasicCanonicalize @ tensors_Plus := BasicCanonicalize /@ tensors;
        BasicCanonicalize @ (num_?NumberQ tensorProduct_) :=
          num BasicCanonicalize @ tensorProduct;
        BasicCanonicalize @ other_ = other;
```

Here is the implementation of the basic algorithm, steps 2 through 7.

```
In[4]:= BasicCanonicalize @ tensorProduct : (head_ @ __Tensor | _Tensor) :=
          Block[{S, signedConfiguration, allEquivalentConfigurations,
              tensorId, tensorIdentifiers, configurationLength, productHead},
            {productHead, tensorIdentifiers, signedConfiguration} =
              encodeTensors @ tensorProduct;
            configurationLength = signedConfiguration[[2]]ten;
            S = Union @ Flatten @ calculateGenerators @ tensorIdentifiers;
            allEquivalentConfigurations = generateConfigurations[signedConfiguration, S];
            If[ tensorIdenticallyZeroQ[allEquivalentConfigurations], 0,
              reconstituteTensors[productHead,
                tensorIdentifiers, Minc @ allEquivalentConfigurations]]]
```

From the already quoted Theorem 6.10.A, appearing in §6.10 *Identically Zero Tensors*, we know that a tensor is identically zero if and only if some signed configuration appears with both a positive and a negative sign in the collection of signed configurations generated by `generateConfigurations`, hence if and only if that set contains more signed configurations than configurations. Thus, for the basic algorithm, we can use the following simple cardinality test to check if a tensor is identically zero.

```
In[5]:= tensorIdenticallyZeroQ @ signedConfigurations_List :=
          Union[sc[[2]] &sc /@ signedConfigurations]ten < signedConfigurationsten
```

The remaining subsections give an overview of the encoding and reconstituting routines, then present some examples, and finally prove that our algorithm in fact returns a canonical tensor.

6.7.2 Specification of Generators

The loading of the tensor canonicalization package defines several standard symmetries for common tensors. However, it is obviously necessary to have the capability to easily define new symmetries for new tensors. We can specify new symmetries for tensors using the *Tensors* package function `DeclareSymmetries`. Our basic canonicalization algorithm will function correctly with any set of generators. Yet since the efficiency of our basic canonicalization algorithm is of the order of $|\mathcal{G} *_c t| * |\mathcal{S}|$, it is transparently obvious that we desire the smallest set of generators $\mathcal{S}$ possible.

`DeclareSymmetries` takes as arguments the name of the tensor, the number of indices of the tensor, and the symmetries the tensor should obey. For instance, the Ricci tensor is denoted by `R` and has two indices which we can interchange, that is, $\mathbf{R}_{i\,j} == \mathbf{R}_{j\,i}$ for any indices `i,j`. We can state this as $\mathbf{R}_{i\,j} = (1 \leftrightarrow 2)_+ * \mathbf{R}_{i\,j} = \mathbf{R}_{j\,i}$. Thus, $(1 \leftrightarrow 2)_+$ is a symmetry of the Ricci tensor. We would enter this symmetry as follows.

```
In[6]:= DeclareSymmetries[R, 2, {(1 ↔ 2)+}]
```

All permutations which are effectively transpositions can be entered as transpositions. All other permutations must be entered as signed permutations. For instance, the Riemann tensor is denoted by R, it has four indices and has the symmetries $\mathbf{R}_{i\,j\,k\,l} == -\mathbf{R}_{j\,i\,k\,l}$, $\mathbf{R}_{i\,j\,k\,l} == -\mathbf{R}_{i\,j\,l\,k}$, and $\mathbf{R}_{i\,j\,k\,l} == \mathbf{R}_{k\,l\,i\,j}$. These symmetries correspond to $(1 \leftrightarrow 2)_-$, $(3 \leftrightarrow 4)_-$, and $\{3, 4, 1, 2\}_+$. We would enter these symmetries for the Riemann tensor as follows.

```
In[7]:= DeclareSymmetries[R, 4, {(1 ↔ 2)-, (3 ↔ 4)-, {3, 4, 1, 2}+}]
```

For a final example, consider the tensor `S` which we will define to be symmetric on all 4 of its indices. We do this as follows.

```
In[8]:= DeclareSymmetries[S, 4, {(1 ↔ 2)+, (2 ↔ 3)+, (3 ↔ 4)+}]
```

After specifying the generators for each individual tensor, we turn to considering the set of generators for a tensor product. For example, what are the generators for $\mathbf{R}^{a\,b\,c\,d}\,\mathbf{R}^{m\,n}{}_{b\,a}\,\mathbf{S}_{d\,c\,n\,m}$? Previously, we mentioned that we can obtain these generators using $\left\langle \mathbf{R}^{a\,b\,c\,d}\,\mathbf{R}^{m\,n}{}_{b\,a}\,\mathbf{S}_{d\,c\,n\,m} \right\rangle_S$. The generators returned are basically just shifted versions of the individual generators for each factor plus some additional generators owing to the ability to swap overall factors.

```
In[9]:= ⟨R^abcd R^mn_ba S_dcnm⟩_S
```

```
Out[9]= {(1 ↔ 2)-, (3 ↔ 4)-, (5 ↔ 6)-, (7 ↔ 8)-, (9 ↔ 10)+, (10 ↔ 11)+,
    (11 ↔ 12)+, {3, 4, 1, 2, 5, 6, 7, 8, 9, 10, 11, 12}+,
    {1, 2, 3, 4, 7, 8, 5, 6, 9, 10, 11, 12}+,
    {5, 6, 7, 8, 1, 2, 3, 4, 9, 10, 11, 12}+}
```

Further commentary on the simple code for the creation of the generating sets from the tensor names will be delayed in order to inclusively handle the small complications arising in the optimized algorithm. In addition, we will be much better placed to understand this code once the breakdown of the symmetries has been covered in §6.8 *Generators and Group Theoretic Underpinnings.*

Later, in §6.11 *The Optimized Algorithm,* to guarantee that the optimized canonicalization algorithm returns valid results, we must use generator sets that fulfill specific criteria. This criteria is defined in §6.8.5 *Jointly Recursively Directional and Extrema Stabilizing (JRDES) Sets.* Happily, for the rest of this section, we will not have any specific need to discuss this complication.

Once we progress onto §6.14 *Linear Symmetries and the Complete Algorithm*, we shall introduce one further type of symmetry to handle symmetries like $\mathbf{R}_{abcd} + \mathbf{R}_{acdb} + \mathbf{R}_{adbc} = 0$. Up until then, however, all the symmetries we shall use can be entered with `DeclareSymmetries` and the appropriate signed transpositions and signed permutations.

6.7.3 Encoding and Reconstituting Tensors

In actuality, the longest pieces of our canonicalization code are those involved in both `encodeTensors` and `reconstituteTensors`. We will not present the code for these routines since it consists mainly of book keeping work with no real insight into the algorithm. For specific details, the interested reader can study the complete code at the end of this thesis in §D *Appendix: Tensor Simplification Code*. However, let us give an overview of the functionality of `encodeTensors` and `reconstituteTensors` and present a few examples of the kind of input and output both routines take and return.

The function `encodeTensors` takes as arguments a tensor product over some head, and a possible option to sort the tensors that defaults to `True`. It returns a list of three answers: the product head, a list of tensor identifiers and an initial signed configuration.

Technical Note: `encodeTensors` can actually take any structure of the form *head*[*Tensor*$_1$,...,*Tensor*$_n$], not just the more typical `Times`[*Tensor*$_1$,...,*Tensor*$_n$]. This is necessary, for instance, when dealing with a product of non-commutative tensorial operators — see §7 *Tensor Calculus, Applications, and Quasi-Spin*.

Let us demonstrate `encodeTensors` on a list of tensors.

```
In[10]:= encodeTensors[{R^{αβγτ}, R^{ν}_{ϵ;βγ}, A^{μ}_{νξ;τ}}, sortTensors → False]

Out[10]= {List, {tensorId[1], tensorId[2], tensorId[3]},
          {α_1^↑, s_{2,7}, s_{3,8}, s_{4,12}, s_{5,10}, ϵ_6^↓, s_{7,2}, s_{8,3}, μ_9^↑, s_{10,5}, ξ_11^↓, s_{12,4}}_+}
```

The data structure *tensorId* is used to store all the particular information on each of the tensors. The structure *tensorId* is dynamic, owing to its creation in a `Block`, and so it can be used with impunity throughout the sub-procedures of our algorithm. Associated with each of the tensor identifiers *tensorId*[*n*] are four values. There is the name of the tensor, the index valence (which consists of the number of normal indices, the number of covariant indices and the number of partial derivatives), the offset to each tensor in the signed configuration, and finally the initial indices (from which we can determine the index classes of the indices in the tensor). For instance, if one were to query *tensorId*, one would find that *tensorId*[2] had the following values:

```
tensorId[2][indexValence] = {2, 2, 0}
tensorId[2][indices] = {ν^+, ϵ^-, β^-, γ^-}
tensorId[2][name] = R
tensorId[2][offset] = 5
```
(6.7.a)

Thus in the above example, a tensor named R starts at position 5 in the signed configuration. It has 2 normal indices (hence is a Ricci tensor), 2 covariant derivatives, no partial derivatives and it uses space time indices.

The function `reconstituteTensors` takes as arguments a list of tensor identifiers that reference the information about the tensors, and a signed configuration. From these constituents, `reconstituteTensors` returns the reconstructed tensor. For instance, we can use the output of the previous example as input for the next example.

```
In[11]:= reconstituteTensors @@ %
```

$$\text{Out[11]= } \left\{\mathbf{R}^{\alpha\beta\gamma\delta},\ \mathbf{R}^{\zeta}{}_{\epsilon;\beta\gamma},\ \mathbf{A}^{\mu}{}_{\xi\xi;\delta}\right\}$$

In this example, since the signed configuration was not manipulated, the reconstituted tensor is, up to relabeling, the same as the input tensor. We can obtain a more typically encountered tensor product by replacing the head by times.

```
In[12]:= Times @@ %
```

$$\text{Out[12]= } \mathbf{A}^{\mu}{}_{\xi\xi;\delta}\,\mathbf{R}^{\alpha\beta\gamma\delta}\,\mathbf{R}^{\zeta}{}_{\epsilon;\beta\gamma}$$

The only part of the basic algorithm left unexplained is the calculation of the generators in step 3. We leave this to §D.5 *Constructing Generators from Primitive Generators*. Next, let us consider some simple examples using the basic algorithm; and after that, let us prove some simple properties of the `BasicCanonicalize` function.

6.7.4 Some Simple Examples

Here is an example given in Portugal's paper [266]. Portugal's algorithm is in Maple, and to perform the following calculation it took in the vicinity of 26 seconds on a Pentium 100.

```
In[13]:= BasicCanonicalize[R^abcd R^mn_ba R_dcnm] // Timing
```

$$\text{Out[13]= } \left\{0.516667\ \text{Second},\ -\mathbf{R}^{abcd}\,\mathbf{R}^{ij}{}_{ab}\,\mathbf{R}_{cdij}\right\}$$

The timing above is for a Power Macintosh G3/300 Mhz. Thus, our algorithm is already quite fast. However, after including optimizations, we will be able to speed our algorithm up by at least another order of magnitude. Moreover, our optimized algorithm scales well.

```
In[14]:= BasicCanonicalize[R^epsbetgamtau R^nualpha_taugam S^mu_nuxi] // Timing
```

$$\text{Out[14]= } \left\{2.38333\ \text{Second},\ -\mathbf{R}^{\alpha\gamma\delta\zeta}\,\mathbf{R}^{\beta\epsilon}{}_{\delta\zeta}\,\mathbf{S}^{\mu}{}_{\gamma\xi}\right\}$$

```
In[15]:= BasicCanonicalize[R^epsbetgamtau R^nualpha_taugam T^mu_nualpha]
```

```
Out[15]= R^{β ∈ α γ}  R^{δ ζ}  T^{μ}
                α γ           δ ζ
```

Our algorithm, as it correctly should, recognizes tensors that are equivalent to zero.

```
In[16]:= BasicCanonicalize[R^{a b}  S^{e}  ]
                          c d      a b f
Out[16]= 0
```

Sometimes the canonical form is in a form that is non-standard. For instance, in the following the covariant derivatives are raised when they could be lowered. In fact, in §6.12.1 *Necessity of Indices with Fixed Elevations*, we will discuss the correct handling of partial derivatives.

```
In[17]:= BasicCanonicalize[R^{α β γ τ}  R^{ν}  A^{μ}  ]
                          ϵ ; β γ       ν ξ , τ
Out[17]= A^{μ}  R^{α δ ζ}  R^{β}
          β ξ ,  γ        ϵ ; δ ζ
```

It is a relatively simple task to add some form of post-canonicalization processing, so long as it is deterministic.

6.7.5 Properties of Canonicalization

Let us now formalize the definition of a canonicalizing function. Similar to Geddes[124], let us introduce the following definitions of "canonical" objects — see also [36].

Definition 6.7.A: A *canonicalizing function* f relative to an equivalence relation $\sim$ on a class of syntactical objects or expressions $\mathcal{E}$ is a computable function $f: \mathcal{E} \to \mathcal{E}$ such that for all $a, b \in \mathcal{E}$, the following properties hold.

(*i*) $a \sim f(a)$

(*ii*) $a \sim b \Rightarrow f(a) \equiv f(b)$

Definition 6.7.B: With respect to a canonicalizing function $f : \mathcal{E} \to \mathcal{E}$, we say that the *canonical form of* $a \in \mathcal{E}$ is $f(a)$, hence we say that $a \in \mathcal{E}$ is in *canonical form*, if and only if $a \equiv f(a)$.

To prove that our function `BasicCanonicalize` is a canonicalizing function, we need the following lemma that any two equivalent signed configurations generate the same collection of signed configurations.

Lemma 6.7.A: $\forall\, a, b \in C$ if $a \sim_{\mathcal{G}} b$ then $\mathcal{G} *_c a = \mathcal{G} *_c b$.

Proof: If $a \sim_{\mathcal{G}} b$ then $\exists\, g \in \mathcal{G}$ such that $a = g *_c b$. Therefore $\mathcal{G} *_c a = \mathcal{G} *_c (g *_c b) = (\mathcal{G} \cdot g) *_c b = \mathcal{G} *_c b$. ■

To reiterate, in essence our tensor canonicalization function just picks the minimum element in the set of configurations equivalent to t, that is, it picks $min_c(\mathcal{G} *_c t)$, given that the initial configuration is t.

Theorem 6.7.A: The function `BasicCanonicalize` defined above is a canonicalizing function on tensor products.

Proof: We want to show that when f is the function `BasicCanonicalize` and $\mathcal{E}$ is the set of tensor product expressions, then $f: \mathcal{E} \to \mathcal{E}$ is a canonicalizing function. Hence, for this f we must show that the properties of Definition 6.7.A hold, namely (*i*) $T = f(T)$ and (*ii*) $T_1 = T_2 \Rightarrow f(T_1) \equiv f(T_2)$.

(*i*) Given $T \in \mathcal{E}$ and its corresponding signed configuration $t \in C$, then either T is identically zero or it isn't. If T is identically zero, then $f(T)$ returns zero, thus $T = 0 \equiv f(T)$. If T is non-zero, then $\mathcal{G} *_c t$ has a unique minimum, $min_c(\mathcal{G} *_c t)$, by Theorem 6.10.A. Furthermore, since $min_c(\mathcal{G} *_c t) \in \mathcal{G} *_c t$ it must be the case that $min_c(\mathcal{G} *_c t) \sim_{\mathcal{G}} t$ and thus $f(T) = T$ by Definition 6.4.C.

(*ii*) Given tensor products T_1 and T_2 such that $T_1 = T_2$, then by Definition 6.4.C, we must have $t_1 \sim_{\mathcal{G}} t_2$. If the tensors T_1 and T_2 are identically zero, which can be determined algorithmically, then $f(T_1)$ and $f(T_2)$ must of course both be zero by the construction of f. So assume the tensors are non-zero. Now, by Lemma 6.7.A, $\mathcal{G} *_c t_1 = \mathcal{G} *_c t_2$, hence again by Theorem 6.10.A, these sets have a unique common minimum. So $min_c(\mathcal{G} *_c t_1) = \min_c(\mathcal{G} *_c t_2)$; hence $min_c(\mathcal{G} *_c t_1) \equiv \min_c(\mathcal{G} *_c t_2)$, since equal configurations are identical. Therefore, the tensors returned by f are exactly the same. Thus, in either case, $f(T_1) \equiv f(T_2)$. ■

It is also a trivial result to show that the canonicalization operator is idempotent.

6.7.6 Overview of Optimizations

There are, of course, several obvious enhancements that can be made to our algorithm. The first thing to note is that most of the time the signed permutation group is decomposable into the internal direct product of the groups that make up the symmetries of the individual tensors. Surprisingly though, to make use of this fact in our algorithm is a non-trivial process. We will not make use of such an enhancement and leave a cursory discussion of it to §C.4 *Canonicalizing in Stages*.

Another obvious candidate for simplification is first to move the free indices into their canonical positions, and then to restrict our consideration to the subgroup of those permutations which do not move the free indices. The idea behind this is that once the free indices are in their optimal positions, any permutation that moves a free index will result in a "larger" configuration — recall §6.6.2 *Definition of the Ordering* — so need not be considered. This simplification, which will be incorporated into our optimized algorithm, is given in §6.11.1

Canonicalization of Free Indices. However, the theory underlying this simplification is developed in §6.8 *Generators and Group Theoretic Underpinnings*.

Undoubtedly, the most complex optimization we will make use of in our optimized canonicalization algorithm involves efficiently handling transpositional symmetries; in particular, this implies that we can handle totally symmetric and totally anti-symmetric tensors extremely efficiently. Intuitively, a completely symmetric object is nice in that we can order the indices inside it in any way that we desire. However, from the basic algorithm in this section, we can see that a symmetric tensor with n indices leads to an $n!$ fold increase in the number of configurations. This is bad. There is, however, an elegant resolution to this matter; and indeed, it turns out that tensors can have their "transpositional" parts treated in extremely efficient ways. This treatment is given in §6.9 *Transpositional Canonicalization*. Unfortunately, transpositional canonicalization makes the theory behind the zero-equivalence question quite a bit harder. Fortunately, the code to determine zero-equivalence is still quite fast and takes only 3 lines of *Mathematica* code. The arguments and theory surrounding zero-equivalence are fully given in §6.10 *Identically Zero Tensors*.

To undertake the discussions of these optimizations, it is first necessary to present some of the underlying group theoretic ideas about the fundamental workings of the overall algorithm and to discuss generators. The next section does exactly this.

6.8 Generators and Group Theoretic Underpinnings

The previous sections have discussed permutation generators in cursory terms only. We have not discussed which generating sets we should use, how efficient is it to use other equivalent generating sets, what tensors can be handled, and many of the other issues concerning generators. For the basic algorithm presented in §6.7 *The Basic Canonicalization Algorithm*, any set of generators is sufficient. However, in order to ensure that the optimizations we will make to the basic algorithm are valid, we need specific properties of our generating sets to be true. In this section we will discuss the kinds of generating sets used in our optimized algorithm, as well as comment on some of the group theoretic underpinnings of our algorithm.

6.8.1 The Classification of Symmetries

Throughout this section we will make frequent use of the computational group theory package GAP[121] since it is specifically designed for computations in group theory (and other algebraic structures). In particular, it will be used to illustrate some of the theory and concepts behind our main algorithm. However, the code for the main algorithm does not require GAP and indeed does not use it.

Instead of GAP, we could alternatively use *Mathematica*. Unfortunately, currently the only major package in *Mathematica* for handling abstract algebra is the Tutorial/Book *Exploring Abstract Algebra in Mathematica* by Hibbard[161]; and this pedagogical tool, although useful, is insufficient for our needs here. So we would first have to develop the specific algebraic routines we need for this section in *Mathematica*. Largely, this undertaking would be redundant since these routines are not needed for our main canonicalization algorithm. In contrast, for the work of this section, we can utilize GAP as is. Other ready-to-use alternatives to GAP that we could have used are Magma[223] or its predecessor Cayley[43, 44], among others.

In this section we will make frequent use of the concepts and notation common in group theory for objects such as group actions, orbits and stabilizers — for instance, see Butler[42], Dixon[93] or Wieldant[336]. The reader should have at least some knowledge of group theory. Unless stated otherwise, Ω will be a finite ordered set of points or positions and $\mathcal{G}$ will be a permutation group acting on Ω, that is, $* : \mathcal{G} \times \Omega \to \Omega$. If $\beta \in \Omega$ and $g \in \mathcal{G}$ then the image of β under g, that is $g * \beta$, is denoted by β^g. Also, the orbit of the point β will be denoted by $\beta^{\mathcal{G}}$, that is, $\beta^{\mathcal{G}} = \{\beta^g \mid g \in \mathcal{G}\}$. The subgroup of $\mathcal{G}$ which *stabilizes* the point β will be denoted by $\mathcal{G}_\beta$, that is, $\mathcal{G}_\beta = \{\sigma \in \mathcal{G} \mid \beta^\sigma = \beta\}$. In addition given $g, h \in \mathcal{G}$ we will denote the *conjugate* $g \cdot h \cdot g^{-1}$ by g^h. Typically in our applications throughout this section $\Omega = \{1, 2, \ldots, n\}$, where n is the number of indices in the tensor product under consideration.

Technical Note: One should note that there is no accepted standard as to exactly how conjugation is defined. This is true even after one settles on whether to denote function application on the left, as is most common, or on the right, as used by some algebraists. The two ways one can define conjugation are (*i*) $g \cdot h \cdot g^{-1}$ and (*ii*) $g^{-1} \cdot h \cdot g$. In this thesis we will use $g \cdot h \cdot g^{-1}$. This essentially corresponds to the convention adopted by Dixon[93], MacLane[216] and Rose[276], and is opposed to the convention used by Butler[42] and Fraleigh[114]. In any case, it is somewhat of a mute point in that the final results using conjugates will be isomorphic under either convention, hence in the end, it is immaterial which convention is chosen.

Given a set of signed permutations, $\mathcal{S}$, corresponding to the symmetries for a tensor T, we know that $\mathcal{S}$ generates a signed permutation group $\mathcal{G}$. As has been mentioned before, the generating set $\mathcal{S}$ for the permutation group $\mathcal{G}$ is not unique. We will find that for our algorithm, it will be necessary to construct an over–complete generating set that is still "small", in a way yet to be defined. This subsection will describe the separate parts that make up our generating sets and will also serve as a background for the other subsections in this section.

It is convenient to divide $\mathcal{S}$ into three parts: the transpositions, the symmetries due to name degeneracy, and finally the other remaining symmetries. The symmetries in the second part, those due to name degeneracy, arise when two or more tensor factors in a tensor product have the same name and the same number and type of indices. For instance, in §6.5.2 *The Algorithm for Generating Configurations*, we illustrated the generation of configurations using the tensor product $\mathbf{R}^{abcd}\,\mathbf{R}^{mn}{}_{ba}\,\mathbf{R}_{dcnm}$. This tensor has three tensor factors, all with the same name, R, and the same number of indices. So it is obvious that we can interchange the overall factors; that is,

$$\mathbf{R}^{abcd}\,\mathbf{R}^{mn}{}_{ba}\,\mathbf{R}_{dcnm} = \mathbf{R}^{mn}{}_{ba}\,\mathbf{R}^{abcd}\,\mathbf{R}_{dcnm} = \mathbf{R}^{mn}{}_{ba}\,\mathbf{R}_{dcnm}\,\mathbf{R}^{abcd} = \mathbf{R}_{dcnm}\,\mathbf{R}^{mn}{}_{ba}\,\mathbf{R}^{abcd} \tag{6.8.a}$$

Any symmetry due to the degeneracy of the tensor names carries over directly from tensor products to their corresponding signed configurations, but of course is only valid when the underlying product operation is multiplication or another commutative operation. For instance, tensor operators do not necessarily commute under "multiplication" — see §7 *Tensor Calculus, Applications, and Quasi-Spin*.

Technical Note: When we are canonicalizing expressions where the overall operation is non-commutative, we simply exclude name degeneracy symmetries from our generating sets. In this way we can canonicalize over operations such as non-commutative-times, etc. For example, see §7 *Tensor Calculus, Applications, and Quasi-Spin*.

The third part of $\mathcal{S}$ consists of those members of $\mathcal{S}$ that are neither transpositions nor name degeneracy symmetries. The author is unaware of any common name in group theory for these symmetries. Therefore we will, for lack of a better name, call them *complex symmetries* or *complex permutations*.

Technical Note: Other equally good names for the complex symmetries would be higher-order symmetries, non-simple symmetries, entangling symmetries, or even intricate symmetries. However, these all conjure up other connotations and so were not chosen.

Given a set of generators $\mathcal{S}$, we will denote the subset of transpositional generators by $\mathcal{S}_{\mathcal{T}}$, the subset of name degeneracy generators by $\mathcal{S}_{\mathcal{D}}$, and the subset of complex symmetry generators by $\mathcal{S}_{\mathcal{X}}$. Thus $\mathcal{S} = \mathcal{S}_{\mathcal{T}} \bigcup \mathcal{S}_{\mathcal{D}} \bigcup \mathcal{S}_{\mathcal{X}}$. Let us symbolize these.

```
In[1]:= Symbolize[S_T]; Symbolize[S_X]; Symbolize[S_D];
```

The mechanism whereby we can obtain the symmetries of a tensor product can return the different types of generators separately. For example,

```
In[2]:= {S_T, S_X, S_D} = ⟨R^{abcd} R^{mn}_{ba} R_{dcnm}⟩_{S_T,X,D}
```

```
Out[2]= {{(1 ↔ 2)_, (3 ↔ 4)_, (5 ↔ 6)_, (7 ↔ 8)_, (9 ↔ 10)_, (11 ↔ 12)_},
          {{3, 4, 1, 2, 5, 6, 7, 8, 9, 10, 11, 12}_+,
            {1, 2, 3, 4, 7, 8, 5, 6, 9, 10, 11, 12}_+,
            {1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 9, 10}_+},
          {{5, 6, 7, 8, 1, 2, 3, 4, 9, 10, 11, 12}_+,
            {1, 2, 3, 4, 9, 10, 11, 12, 5, 6, 7, 8}_+}}
```

The subscripts on the $\mathcal{S}$ in $\langle ... \rangle_{\mathcal{S}_{subscripts}}$ indicate which symmetries should be returned and in what order.

It should also be mentioned that some tensors admit symmetries which are linear in nature. For instance, the Riemann tensor, in addition to the aforementioned symmetries, has the linear symmetry $\mathbf{R}_{abcd} + \mathbf{R}_{acdb} + \mathbf{R}_{adbc} = 0$. Similar to this linear symmetry, are the celebrated Bianchi identities, $\mathbf{R}_{\alpha\beta\gamma\delta;\sigma} + \mathbf{R}_{\alpha\beta\delta\sigma;\gamma} + \mathbf{R}_{\alpha\beta\sigma\gamma;\delta} = 0$. Neither of these linear symmetries can be viewed as a result of our group action operating on a configuration because new terms are possibly being added. We will return to linear symmetries in due course in §6.14 *Linear Symmetries and the Complete Algorithm*, but for now they lie outside the scope of our algorithm.

6.8.2 Examining the Classification of Symmetries via GAP

In this subsection we use GAP to examine the separate parts of the generating set that we introduced in the previous subsection, namely, the transpositional generators, the name degeneracy generators, and the complex generators. Let us begin by re-examining the generators of the permutation group used by our algorithm to represent the symmetries of the tensor product $\mathbf{R}^{abcd}\,\mathbf{R}^{mn}\,\mathbf{R}_{ba}\,\mathbf{R}_{dcnm}$.

```
In[3]:= ⟨R^abcd R^mn R_ba R_dcnm⟩_S

Out[3]= {(1 ↔ 2)_, (3 ↔ 4)_, (5 ↔ 6)_, (7 ↔ 8)_, (9 ↔ 10)_,
    (11 ↔ 12)_, {3, 4, 1, 2, 5, 6, 7, 8, 9, 10, 11, 12}_+,
    {1, 2, 3, 4, 7, 8, 5, 6, 9, 10, 11, 12}_+,
    {1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 9, 10}_+,
    {5, 6, 7, 8, 1, 2, 3, 4, 9, 10, 11, 12}_+,
    {1, 2, 3, 4, 9, 10, 11, 12, 5, 6, 7, 8}_+}
```

Observe that this set of generators is over–complete. Let us use GAP to quickly show this. First though, a note on GAP notation. GAP represents permutations in disjoint cycle notation, as well as denoting assignments by := and semantic equality tests by = . Also GAP represents lists in the form $[e_1, e_2, \ldots, e_n]$, as do other symbolic computation languages such as Maple[244, 329], MuPad[118, 289] and AXIOM[177], as well as some modern functional languages like ML[146], Haskell[89] and Clean[264, 265]. For now we will ignore the sign of a permutation and comment on this omission in the next subsection. First, let us enter the generators into GAP in disjoint cycle notation. Since GAP has no capabilities for working with mathematical notations, let us respectively use **ST**, **SX**, **SD** instead of $\mathcal{S}_{\mathcal{T}}$, $\mathcal{S}_{\mathcal{X}}$, $\mathcal{S}_{\mathcal{T}}$.

```
gap> ST := [(1,2), (3,4), (5,6), (7,8), (9,10), (11,12)];
[ (1,2), (3,4), (5,6), (7,8), ( 9,10), (11,12) ]
gap> SX := [(1,3)(2,4),(5,7)(6,8),(9,11)(10,12)];
[ (1,3)(2,4), (5,7)(6,8), ( 9,11)(10,12) ]
gap> SD := [(1,5)(2,6)(3,7)(4,8), (5,9)(6,10)(7,11)(8,12)];
[ (1,5)(2,6)(3,7)(4,8), ( 5, 9)( 6,10)( 7,11)( 8,12) ]
```

It is now easy to show that the group generated by $\mathcal{S}_{\mathcal{T}} \bigcup \mathcal{S}_{\mathcal{X}} \bigcup \mathcal{S}_{\mathcal{D}}$ is the same as the group generated by $\{(1, 2), (1, 3)(2, 4)\} \bigcup \mathcal{S}_{\mathcal{D}}$.

```
gap> G := Group( Union(ST,SX,SD) );
<permutation group with 11 generators>
gap> G = Group( Union ( [(1,2), (1,3)(2,4)], SD));
true
```

Therefore, our generating set is obviously over–complete.

But for the existence of the name degeneracy symmetries, the whole permutation group would nicely factorize into the internal direct product of the permutation groups $\mathcal{R}_G$ of the constituent tensors R, that is, $\mathcal{G} \simeq \mathcal{R}_G \times \mathcal{R}_G \times \mathcal{R}_G$. Unfortunately, the name degeneracies cause this factorization to be erroneous.

```
#RG denotes the symmetry group of the Riemann tensor
gap> RG := Group( (1,2), (3,4), (1,3)(2,4) );
Group([ (1,2), (3,4), (1,3)(2,4) ])
gap> RGxRGxRG := DirectProduct(RG,RG,RG);
<permutation group with 9 generators>
gap> RGxRGxRG = G;
false
gap> RGxRGxRG = Group( Union(ST,SX) );
true
```

Of course there are techniques in computational group theory whereby one can extend a group by a set of generators (for instance the GAP function **GroupClosure**). In fact, it is easy to show that extending the direct product group $\mathcal{R}_G \times \mathcal{R}_G \times \mathcal{R}_G$ by closure with the name degeneracy generators in $\mathcal{S}_{\mathcal{D}}$ leads to an exact copy of $\mathcal{G}$.

```
gap> ClosureGroup(RxRxR, SD) = G;
true
```

Although it is possible to precompute $\mathcal{R}_G$, trying to use the precomputed results raises complications for our optimized algorithm. Furthermore, precomputation does not save that much time since, as we commented before, our main focus is not in calculating $\mathcal{G}$ but in calculating $\mathcal{G} *_c t$. For instance, as you may recall from §6.5.3 *The Number of Configurations*, $|\mathcal{G}| = 3072$ while $|\mathcal{G} *_c t| = 64$. We can also verify these sizes using GAP.

```
gap> Size(G);
3072
```

To verify that the orbit $\mathcal{G} *_c t$ has 64 elements using GAP is slightly more complicated. GAP's facilities can be used to calculate an orbit of a configuration if one chooses an appropriate representation for configurations and an appropriate group action. If we use the action of "on sets of sets", which GAP names `OnSetsSets`, and make the identification of an $s_{i,j}$ label with the list $[i,j]$, then we can for instance generate the orbit of configurations equivalent to the configuration for $\mathbf{R}^{abcd}\,\mathbf{R}^{mn}{}_{ba}\,\mathbf{R}_{dcnm}$, namely the configuration $\{s_{1,8}, s_{2,7}, s_{3,10}, s_{4,9}, s_{5,12}, s_{6,11}, s_{7,2}, s_{8,1}, s_{9,4}, s_{10,3}, s_{11,6}, s_{12,5}\}$.

```
gap> initialConfig := [[1,8],[2,7],[3,10],[4,9],[5,12],[6,11]];
[ [ 1, 8 ], [ 2, 7 ], [ 3, 10 ], [ 4, 9 ], [ 5, 12 ], [ 6, 11 ] ]
gap> Gorbit := Orbit(G, initialConfig, OnSetsSets);;
gap> Size(Gorbit);
64
```

6.8.3 GAP, Mathematica, and Tensor Simplification

In the light of the above computations in GAP, a couple of questions arise. First, we note that we used unsigned configurations and unsigned permutations. Are the results we obtained essentially the same as we would get performing the calculations on signed configurations with signed permutations? In particular, is the size of $\mathcal{G}$ the same? As first noted in §6.6.4 *Almost Total Orderings* and as later shown in §6.10.1 *Zero Equivalence in the Basic Algorithm*, in the set of equivalent signed configurations the same configuration can be repeated with a different sign if and only if the tensor being canonicalized is identically zero. In our example using $\mathbf{R}^{abcd}\mathbf{R}^{mn}{}_{ba}\mathbf{R}_{dcnm}$, the tensor is non-zero, so the unsigned version of the permutation group $\mathcal{G}$ is isomorphic to the signed version of the permutation group $\mathcal{G}$.

Second, from the ease with which we calculated the orbit $\mathcal{G} *_c t$ in the previous subsection, §6.8.2, one might get the impression that it would be very easy to implement the essential core calculations of our canonicalization algorithm in GAP. So why have we bothered to implement our algorithm in *Mathematica*? Actually, although we can achieve the same results in GAP, it takes much more work than the code skeleton given in the previous subsection, §6.8.2. For instance, how would we include free indices? We could change the default action of `OnSetSets` to a new user defined action which we would have to define and implement. Alternatively, we could include "virtual" positions in our configurations which are not moved by the permutation group $\mathcal{G}$. Each of these virtual positions would correspond to a free index; and since such a virtual position is not moved by the action of the permutations of $\mathcal{G}$, the labels remain fixed throughout the orbit of equivalent configurations. For an example involving virtual positions, consider the configuration obtained from the previous configuration by putting free indices in positions 1, 5, 6, 8, 11, 12. Our configuration has 12 positions, so we link position 1 with the virtual position 13, 5 with 14, etc.

```
gap> initialConfig:=[[1,13],[2,7],[3,10],[4,9],[5,14],[6,15],[8,16],[11,17],[12,18]];;
gap> Gorbit:=Orbit(G, initialConfig, OnSetsSets);;time;
1139
gap> Size(Gorbit);
1536
```

In GAP timings are measured in milliseconds, so finding the above orbit of 1536 equivalent configurations took approximately a second. This of course is faster than *Mathematica* could do such a calculation, but this is hardly surprising since it is highly likely that GAP is internally optimized for such calculations. However, speed is not the overall determining factor. If it were, then a low-level procedural version of our algorithm, say implemented in C++ or Pascal, would be the preferred choice. Besides, in our optimized algorithm we will not have to calculate the orbit of an initial configuration. Also, it would be no easier in GAP to create the routine that compares signed configurations, nor the routine that generates the degeneracy symmetries, nor the routine that generates the configurations for a product tensor from the

configurations for the individual tensors making up the product, nor many of the other routines that would make up our algorithm or a corresponding one if we chose to use GAP. Probably the best answer is that, to the author's knowledge, tensor analysis is rarely done in GAP, if at all, so implementing a tensor optimization algorithm in GAP would be of arguable use. Besides, in GAP one could not easily do any of the calculus aspects of tensor analysis, which are of course vital.

The above arguments should not detract from the GAP language in any way. It appears to be a well implemented and extremely useful language for group theory and other abstract algebra investigations.

Actually as a side note, the desire to link *Mathematica* and GAP provides another perfect example of the value of the *Notation* package, developed in this thesis. Using *Mathlink* to link *Mathematica* with GAP would allow all users of *Mathematica* access to the specialized group theoretic algorithms and tools bundled with the GAP language. The interface to the GAP system is relatively primitive, text based, and command line driven, so it would appear to be an ideal candidate for hooking up to *Mathematica* via the *Mathematica* Mathlink protocol. Such a pairing would also be highly beneficial for GAP since then, using the *Notation* package, all the technical group theoretic notation common in group theory could be introduced and then used as valid *Mathematica* input. For instance, it should be quite possible to define semantics for input like $(G \times R \to G/T)(K) \lhd H$. Such a development of group-theoretic notation in *Mathematica* would exactly parallel the development of the notation common to physics that has been implemented in this thesis, like the notation for tensors in §3.4 *Tensorial Notation* and for Bras and Kets in §2.7 *Example: Bra-Ket Notation*, etc.

This subsection, together with the previous subsection, has served to reinvestigate, in the language of GAP, some of the familiar concepts first presented in §6.5.3 *The Number of Configurations*. This showcasing of GAP has hopefully given the reader the benefit of a cursory exposure to the language of GAP, and in addition has also underscored the concepts in a group theoretic manner such that the reader can see the rudimentary beginnings of an implementation of our basic canonicalization algorithm in GAP.

6.8.4 Generating Sets

As is well known, we can easily find a set of generators for any group $\mathcal{G}$. A simple method is as follows: initially set S to any element of $\mathcal{G}$ and then successively add a new element to S from $\mathcal{G} \setminus \langle S \rangle$ until $\langle S \rangle = \mathcal{G}$. This is essentially the same sort of technique as the sieve of Eratosthenes which finds all the prime numbers in the range $1, \ldots, m$. However, unless we carefully pick which generators we sieve on at each stage, we cannot guarantee any properties of our generating set, such as minimum cardinality of the generating set or "minimum size" of each generator, etc. For instance, in cycle notation both of the following sets generate the same group.

```
gap> Group((1,2),(1,2,3,4,5,6)) = Group((1,2),(2,3),(3,4),(4,5),(5,6));
true
```

Clearly, the first set of generators is smaller while in the second set each generator is simpler.

Not surprisingly, computational group theorists have already tackled the question of what is the "best" set of generators for a group. Authors like Butler[42], Dixon[93], and languages like GAP, Magma[223], all give routines for creating generating sets with specific properties. Unfortunately, none of these routines correspond to exactly what we need, but these routines are a starting point from which we can adapt algorithms in order to create generating sets with the specific properties that we need.

Two particular notions which have frequent application in computational group theory are that of a base and that of a strong generating set. A *base* for a group $\mathcal{G}$ is a sequence of points in Ω, say $\beta_1, \beta_2, \ldots, \beta_n$, such that the corresponding stabilized subgroup chain terminates in the identity subgroup, that is,

$$\mathcal{G} \geqslant \mathcal{G}_{\beta_1} \geqslant \mathcal{G}_{\beta_1,\beta_2} \geqslant \ldots \geqslant \mathcal{G}_{\beta_1,\beta_2,\ldots\beta_n} = \{e\} \tag{6.8.b}$$

A set of generators $\mathcal{S}$ for $\mathcal{G}$ is said to be *strong* relative to the base $\beta_1, \ldots, \beta_n$ if $\mathcal{S}_{\beta_1,\ldots,\beta_m} = \{\sigma \in \mathcal{S} \mid \beta_i^\sigma = \beta_i \text{ for } 1 \leq i \leq m\}$ generates $\mathcal{G}_{\beta_1,\ldots,\beta_m}$, for $1 \leq m \leq n$. These concepts were first developed by Sims[296, 297]. Descriptions of algorithms to calculate strong generating sets for a given base are presented in Butler[42]. We can demonstrate this in GAP by finding a strong generating set for the group $\mathcal{G}$ considered in the previous subsection, §6.8.3, with the base [1 ...12].

```
gap> StrongGeneratorsStabChain(MinimalStabChain(G));
[ (11,12), ( 9,10), ( 9,11)(10,12), ( 7, 8), ( 5, 6), ( 5, 7)( 6, 8), ( 5, 9)( 6,10)( 7,11)( 8,12), ( 3,
4), ( 1, 2), ( 1, 3)( 2, 4), ( 1, 9, 5)( 2,10, 6)( 3,11, 7)( 4,12, 8) ]
```

Unfortunately, strong generating sets are not exactly what we need since we do not know what order the base points are going to be in. However, we will need a similar concept to achieve our first optimization. In §6.11.1 *Canonicalization of Free Indices* we will present the algorithm that will move the free high and low indices into their canonical positions and then find generators for the stabilized subgroup of permutations that do not move the free indices. This optimization again reduces the number of configurations that have to be considered. We formalize our specific needs in the following subsection.

6.8.5 Jointly Recursively Directional and Extrema Stabilizing (JRDES) Sets

As stated earlier, we assume that our group of permutations $\mathcal{G}$ acts on a set of points $\Omega = \{1, \ldots, n\}$ with the usual ordering. Hence we can speak of one position or point of Ω being to the left or right of another, and thus speak of the maximum and the minimum points of an orbit. We need a set of generators for $\mathcal{G}$ such that any point β in Ω can be moved to the left and to the right as far as its orbit, $\beta^{\mathcal{G}}$, allows. That is, we need to be able to transform β to $min(\beta^{\mathcal{G}})$ and also to $max(\beta^{\mathcal{G}})$. It is important that we can do this in a "downhill" method with just the generators alone rather than the full group since the size of the group can be extremely large whereas the size of the generating set is generally much smaller.

Definition 6.8.A: We say that $\mathcal{S}$ is a *directional* set of generators for a permutation group $\mathcal{G}$ acting on the set of points Ω if

(*i*) $\mathcal{S}$ generates $\mathcal{G}$, that is, $\mathcal{G} = \langle \mathcal{S} \rangle$

(*ii*) for each $\beta \in \Omega$ such that $\beta \neq max(\beta^{\mathcal{G}})$, there exists a $\sigma \in \mathcal{S}$ such that $\beta^{\sigma} > \beta$

(*iii*) for each $\beta \in \Omega$ such that $\beta \neq min(\beta^{\mathcal{G}})$, there exists a $\sigma \in \mathcal{S}$ such that $\beta^{\sigma} < \beta$

We also need to be able to find stabilized subgroups with a minimum of effort. The points on which we need to be able to stabilize are the extrema orbit points, that is, the points that are either a minimum or a maximum point of an orbit of a point in Ω under $\mathcal{G}$. To this end we define the following concept.

Definition 6.8.B: The *orbit extrema* of a set of points Ω under a permutation group $\mathcal{G}$ are the extrema of the orbits of the points in Ω. Denoting the set of orbit extrema by $Ext(\mathcal{G}, \Omega)$, we formally have $Ext(\mathcal{G}, \Omega) = \bigcup_{\gamma \in \Omega} \{min(\gamma^{\mathcal{G}}), max(\gamma^{\mathcal{G}})\}$.

Besides the obvious "intuitive" extremum points, one should note that any point that is not moved is also an extremum. If our generating set allows us to easily find subgroups stabilized on the orbit extrema, we give it the special designation of being *extrema stabilizing*.

Definition 6.8.C: $\mathcal{S}$ is an *extrema stabilizing set of generators* for a group $\mathcal{G}$ which acts on Ω if

(*i*) $\mathcal{S}$ generates $\mathcal{G}$, that is, $\mathcal{G} = \langle \mathcal{S} \rangle$

(*ii*) for each $\beta \in Ext(\mathcal{G}, \Omega)$ the set $\mathcal{S}_\beta = \{\sigma \in \mathcal{S} \mid \beta^{\sigma} = \beta\}$ generates $\mathcal{G}_\beta$, that is $\mathcal{G}_\beta = \langle \mathcal{S}_\beta \rangle$

Finally when we actually use a set of generators in §6.11.1 *Canonicalization of Free Indices* it transpires that we need it to be both extrema stabilizing and directional. Furthermore, we need both of these properties to be recursive.

Definition 6.8.D: A set of generators $\mathcal{S}$ for a permutation group $\mathcal{G}$ is *jointly recursively directional and extrema stabilizing* (JRDES) if

(*i*) $\mathcal{S}$ is a directional set of generators for $\mathcal{G}$
(*ii*) $\mathcal{S}$ is an extrema stabilizing set of generators for $\mathcal{G}$
(*iii*) for each $\beta \in Ext(\mathcal{G}, \Omega)$ the set of generators $\mathcal{S}_\beta$ that generate $\mathcal{G}_\beta$, given by Definition 6.8.C, is again jointly recursively directional and extrema stabilizing

By using generator sets satisfying the JRDES property, as compared to the more standard strong generating set for a base, we have traded possible minimality of the generating set for the ability to choose any sequence of orbit extrema to stabilize on.

The question of existence of a set of generators that is JRDES is easy to answer: the whole group is such a set of generators. Actually we can easily give a much better answer: we could use the union of all strong generating sets for all permutations of the base $[1 \ldots n]$. This resulting set would obviously satisfy the requirements of being JRDES.

Sometimes we will talk about a set of generators with specific properties without reference to the group. This is of course permissible because the set of generators implicitly specifies the group.

In most cases we will be able to find a "small" set of generators for our tensors which is JRDES. Fortuitously, the tensor product operation creates a new permutation supergroup which is also "nicely" JRDES generated if each of the groups corresponding to the tensors comprising the tensor product is "nicely" JRDES generated. The next theorem formalizes this. However, first for reference, the reader should recall from basic group theory the result that $\mathcal{G}$ is the internal direct product of subgroups $\mathcal{H}$ and $\mathcal{K}$ if and only if (*i*) $\mathcal{G} = \mathcal{HK}$, (*ii*) $hk = kh$ for all $h \in \mathcal{H}$ and $k \in \mathcal{K}$, and (*iii*) $\mathcal{H} \cap \mathcal{K} = \{e\}$. (See Fraleigh[114], MacLane[216].)

Theorem 6.8.A: Assume that the signed permutation group $\mathcal{G}$ which acts on Ω is the internal direct product of two subgroups, $\mathcal{H}$ and $\mathcal{K}$, that is $\mathcal{G} = \mathcal{HK}$. Also assume that both $\mathcal{H}$ and $\mathcal{K}$ have JRDES generating sets, say $\mathcal{S}$ and $\mathcal{V}$. If the non-trivial orbits of $\mathcal{H}$ and $\mathcal{K}$ do not overlap, then $\mathcal{G}$ is JRDES generated by $\mathcal{S} \bigcup \mathcal{V}$.

Proof: $\mathcal{G} = \mathcal{HK} = \langle \mathcal{S} \rangle \langle \mathcal{V} \rangle$ and $hk = kh$ for all $h \in \mathcal{H}$ and $k \in \mathcal{K}$ since $\mathcal{G}$ is the internal direct product of $\mathcal{H}$ and $\mathcal{K}$. It follows that $\langle \mathcal{S} \bigcup \mathcal{V} \rangle = \langle \mathcal{S} \rangle \langle \mathcal{V} \rangle = \mathcal{G}$, hence $\mathcal{S} \bigcup \mathcal{V}$ generates $\mathcal{G}$.

We next show that $\mathcal{S} \bigcup \mathcal{V}$ is a directional generating set. Consider any $\beta \in \Omega$ such that $\beta \neq min(\beta^{\mathcal{G}})$. Then obviously the orbit of β is non-trivial under $\mathcal{G}$. This means that β is moved by some element in either $\mathcal{H}$ or $\mathcal{K}$. This implies that either the orbit $\beta^{\mathcal{H}}$ or the orbit $\beta^{\mathcal{K}}$ is non–trivial. (They cannot both be trivial since then $\beta^{\mathcal{G}} = \beta^{\mathcal{HK}} = \{\beta\}$ but this would contradict the fact that $\beta^{\mathcal{G}}$ is non-trivial; furthermore they cannot both be non-trivial for this would contradict our assumption of non-overlapping non-trivial orbits.) Therefore either $\beta^{\mathcal{G}} = \beta^{\mathcal{H}}$ or $\beta^{\mathcal{G}} = \beta^{\mathcal{K}}$, but since $\mathcal{H}$ and $\mathcal{K}$ are generated by $\mathcal{S}$ and $\mathcal{V}$, which are both JRDES, it must be the case that there exists $\sigma \in \mathcal{S} \bigcup \mathcal{V}$ such that $\beta^\sigma < \beta$. Similarly if $\beta \neq max(\beta^{\mathcal{G}})$ we can find $\sigma \in \mathcal{S} \bigcup \mathcal{V}$ such that $\beta^\sigma > \beta$. Thus the set $\mathcal{S} \bigcup \mathcal{V}$ forms a directional generating set for $\mathcal{G}$.

Let us now show the extrema stabilizing criteria. Consider any $\beta \in Ext(\mathcal{G}, \Omega)$, that is, β which is an orbit extremum of $\mathcal{G}$. It follows that $\beta \in Ext(\mathcal{H}, \Omega)$ and $\beta \in Ext(\mathcal{K}, \Omega)$; for if not then β could be moved both left and right by $\mathcal{H}$ or by $\mathcal{K}$, hence obviously it can moved both left and right by $\mathcal{G}$, contradicting $\beta \in Ext(\mathcal{G}, \Omega)$. Therefore $\mathcal{H}_\beta$ and $\mathcal{K}_\beta$ are generated by $\mathcal{S}_\beta$ and $\mathcal{V}_\beta$ respectively. Furthermore, $\mathcal{H}_\beta \bigcap \mathcal{K}_\beta \subseteq \mathcal{H} \bigcap \mathcal{K} = \{e\}$. Also, since the elements of $\mathcal{H}$ and $\mathcal{K}$ commute, then unquestionably the elements of $\mathcal{H}_\beta$ and $\mathcal{K}_\beta$ must commute. Therefore, $\mathcal{H}_\beta \mathcal{K}_\beta$ forms an internal direct product subgroup of $\mathcal{G}$.

Let us now show that the subgroup $\mathcal{H}_\beta \mathcal{K}_\beta$ is in fact $\mathcal{G}_\beta$. Consider any $g \in \mathcal{G}_\beta$. Necessarily $\beta^g = \beta$. Since $\mathcal{G}$ is the internal direct product of $\mathcal{H}$ and $\mathcal{K}$, there must exist unique $h \in \mathcal{H}$ and $k \in \mathcal{K}$ such that $g = h\,k$, hence $\beta = \beta^g = \beta^{hk} = (\beta^h)^k$. If $\beta^h = \gamma \neq \beta$ then clearly $\gamma^k = \beta$ and hence $\beta^{k^{-1}} = \gamma$, but this implies that both $\beta^{\mathcal{H}}$ and $\beta^{\mathcal{K}}$ are non-trivial; and since these orbits both contain at least β they unquestionably overlap, thus contradicting the initial assumption. Consequently, $\beta^h = \beta$ and hence $\beta^k = \beta$, which means that $h \in \mathcal{H}_\beta$ and $k \in \mathcal{K}_\beta$. Therefore $\mathcal{G}_\beta \subseteq \mathcal{H}_\beta \mathcal{K}_\beta$. It is easy to verify that $\mathcal{G}_\beta \supseteq \mathcal{H}_\beta \mathcal{K}_\beta$ and therefore $\mathcal{G}_\beta = \mathcal{H}_\beta \mathcal{K}_\beta$. Since $\mathcal{H}_\beta$ and $\mathcal{K}_\beta$ are generated by $\mathcal{S}_\beta$ and $\mathcal{V}_\beta$ it follows that $\mathcal{G}_\beta$ is generated by $\mathcal{S}_\beta \bigcup \mathcal{V}_\beta = (\mathcal{S} \bigcup \mathcal{V})_\beta$.

We have now shown that $\mathcal{S} \bigcup \mathcal{V}$ is a directional and extrema stabilizing set of generators for $\mathcal{G}$. To show both of these properties are jointly recursively true it is easy to note that the conditions and circumstances of the theorem are still true for the stabilized subgroups. Specifically, the non-trivial orbits of $\mathcal{H}_\beta$ and $\mathcal{K}_\beta$ do not overlap, $\mathcal{G}_\beta$ is the internal direct product of $\mathcal{H}_\beta$ and $\mathcal{K}_\beta$, and furthermore both of these subgroups are respectively JRDES generated by $\mathcal{S}_\beta$ and $\mathcal{V}_\beta$. Therefore we can again apply the above arguments to stabilize on a new point γ and so on. This of course can be repeated recursively and hence we can conclude that $\mathcal{S} \bigcup \mathcal{V}$ is a JRDES set of generators for $\mathcal{G}$. ■

Even though we have only stated and proved this theorem for two groups, it is obvious that it generalizes to a product of n groups which are each JRDES generated and whose non–trivial orbits do not overlap.

Actually, the above theorem can be made more inclusive, but in its present form it is sufficient for our needs. However, it should also be pointed out that just being able to find an $\mathcal{H}$ and $\mathcal{K}$ which are JRDES generated such that $\mathcal{G} = \mathcal{H} \times_{interal} \mathcal{K}$ is not always sufficient to guarantee that $\mathcal{G}$ can be JRDES generated with the generators of $\mathcal{H}$ and $\mathcal{K}$. To guarantee that $\mathcal{G}$ is JRDES generated by the generators of $\mathcal{H}$ and $\mathcal{K}$ we need the property that the non-trivial orbits cannot overlap. A simple counterexample when the orbits overlap is provided by the permutations $\rho = (1, 4)(2, 3)$ and $\sigma = (1, 3)(2, 4)$. The point 3 is an extremum of the orbits under the groups $\langle\rho\rangle$ and $\langle\sigma\rangle$, yet the group generated by ρ *and* σ moves 3 to 4. Therefore, the generators ρ and σ do not have the required directional property that they can move a point towards its orbit maximum, in this case 3 to 4. Yet, $\langle\rho\rangle$ and $\langle\sigma\rangle$ are both subgroups of $\langle\rho,\sigma\rangle$ and their internal direct product forms the whole group $\langle\rho,\sigma\rangle$. Hence $\langle\rho,\sigma\rangle$ is not JRDES generated by ρ and σ.

The restriction in Theorem 6.8.A of $\mathcal{G}$ being the internal direct product of two subgroups means that we cannot apply that result to the tensor product of two versions of a tensor, for example

the tensor product of two Riemann tensors. This is because there are degeneracy generators associated with the later product, connecting the two subgroups. This complication is resolved by Theorem 6.8.E in the concluding subsection §6.8.8 *Tensor Products and their JRDES Generators.*

The next subsections will consider specific sets of generators that are jointly recursively directional and extrema stabilizing (JRDES).

6.8.6 Adjacent Transpositions

The simplest sets of generators that are jointly recursively directional and extrema stabilizing (JRDES) are sets of adjacent transpositions.

Definition 6.8.E: A transposition τ is said to be *adjacent* if it is of the form $(i \leftrightarrow i+1)$ for some i.

Let us consider a few examples of sets consisting entirely of adjacent transpositions and also sets containing non-adjacent of transpositions as well. The set $\{(1 \leftrightarrow 2), (2 \leftrightarrow 3), (3 \leftrightarrow 4), (7 \leftrightarrow 8), (8 \leftrightarrow 9)\}$ clearly consists of just adjacent transpositions. The transpositions in $\{(1 \leftrightarrow 2), (1 \leftrightarrow 3)\}$ are obviously not all adjacent but this set generates the same group as does the adjacent set $\{(1 \leftrightarrow 2), (2 \leftrightarrow 3)\}$. Lastly, the set $\{(1 \leftrightarrow 3), (3 \leftrightarrow 4)\}$ is fundamentally non-adjacent, that is, it does not generate a group which can be generated by any set of adjacent transpositions.

Definition 6.8.F: A group $\mathcal{G}$ is said to be *adjacently generated* if it can be generated by a set of adjacent transpositions $\mathcal{T} \subseteq \mathcal{G}$.

Let us now prove that a set of adjacent transpositions is a JRDES generating set.

Theorem 6.8.B: Every set of adjacent transpositions $\mathcal{T}$ is a jointly recursively directional and extrema stabilizing (JRDES) set of generators for the group $\langle \mathcal{T} \rangle$.

Proof: Consider a point $\beta \in \Omega$ having a non-trivial orbit under the group generated by the set of adjacent transpositions $\mathcal{T}$. The non-trivial orbit $\beta^{\langle \mathcal{T} \rangle}$ must be connected since the transpositions are adjacent. Furthermore, the set $\mathcal{A} = \{(\beta_{min} \leftrightarrow \beta_{min}+1), (\beta_{min}+1 \leftrightarrow \beta_{min}+2), \ldots, (\beta_{max}-1 \leftrightarrow \beta_{max})\}$, where $\beta_{min} = min(\beta^{\langle \mathcal{T} \rangle})$ and $\beta_{max} = max(\beta^{\langle \mathcal{T} \rangle})$, must be a subset of $\mathcal{T}$. We will first show that this set $\mathcal{A}$ is JRDES and then deal with the whole of $\mathcal{T}$.

First, to show that the set of transpositions $\mathcal{A}$ is directional consider any point γ in $\{\beta_{min}, \ldots, \beta_{max}\}$. Intuitively, we can move γ to the right and/or to the left by an element in $\mathcal{A}$. Formally, if $\gamma \in \{\beta_{min}, \ldots, \beta_{max}-1\}$ then we can move γ to the right by $(\gamma \leftrightarrow \gamma+1) \in \mathcal{A}$ since $\gamma^{(\gamma \leftrightarrow \gamma+1)} = \gamma+1 > \gamma$. Conversely, if $\gamma \in \{\beta_{min}+1, \ldots, \beta_{max}\}$ then we can similarly move γ to the left by $(\gamma-1 \leftrightarrow \gamma) \in \mathcal{A}$.

Second, to show that $\mathcal{A}$ is an extrema stabilizing set of generators we need to consider stabilizing $\mathcal{G} = \langle \mathcal{A} \rangle$ on β_{min} as well as stabilizing $\mathcal{G}$ on β_{max}. For the β_{min}

case, it should be obvious that $\mathcal{G}_{\beta_{min}}$ is generated by $\{(\beta_{min}+1 \leftrightarrow \beta_{min}+2), \ldots, (\beta_{max}-1 \leftrightarrow \beta_{max})\}$ which is indeed $\mathcal{A}_{\beta_{min}}$, as desired. The β_{max} case is analogous.

Finally, both properties must be jointly true for any extrema stabilized subgroup of $\mathcal{G}_{\beta_{min}}$ or $\mathcal{G}_{\beta_{max}}$, since such a subgroup will, by the argument above, be generated by a smaller set of adjacent transpositions, and so on. Thus the set of transpositions $\mathcal{A}$ is a JRDES set of generators for $\langle\mathcal{A}\rangle$.

Let us return to the issue of the whole group $\langle\mathcal{T}\rangle$. We can naturally partition $\mathcal{T}$ into subsets $\mathcal{T}_i$ that are just the equivalence classes of the equivalence relation $\sim^*$ defined on transpositions by $(\beta \leftrightarrow \beta+1) \sim^* (\gamma \leftrightarrow \gamma+1)$ if and only if $\beta^{\langle\mathcal{T}\rangle} = \gamma^{\langle\mathcal{T}\rangle}$. These subsets $\mathcal{T}_i$ of course generate subgroups $\langle\mathcal{T}_i\rangle$ which form the factors for an internal direct product group equal to $\langle\mathcal{T}\rangle$. It is easy to show for any pair of these subgroups, that they only have the identity in common and their non-trivial orbits do not overlap. Furthermore, the internal direct product of all of the subgroups $\langle\mathcal{T}_i\rangle$ generated by the various equivalence classes is equal to $\langle\mathcal{T}\rangle$. Since we know, by the above arguments for a single connected subset, that each one of these subgroups $\langle\mathcal{T}_i\rangle$ is JRDES generated by $\mathcal{T}_i$, then by Theorem 6.8.A it must follow that $\langle\mathcal{T}\rangle$ is JRDES generated by $\mathcal{T}$. ■

The following theorem, which we will need later, describes the cycle structure of a conjugated cycle. MacLane[216] presents a beautifully simple proof of this theorem, and we include it almost verbatim here.

Theorem 6.8.C: If $\gamma \in S_n$ is the cycle $(i_1, i_2, \ldots, i_m)$ then $\sigma\cdot\gamma\cdot\sigma^{-1} = (\sigma(i_1), \sigma(i_2), \ldots, \sigma(i_m))$.

Proof: Let $\sigma\cdot\gamma\cdot\sigma^{-1}$ act on any positive integer $j \leq n$. Clearly, $j = \sigma(\sigma^{-1}(j))$. If $i = \sigma^{-1}(j)$ is not one of the i_k, then the action of $\sigma\cdot\gamma\cdot\sigma^{-1}$ on j is $\sigma(i) \overset{\sigma^{-1}}{\rightarrow} i \overset{\gamma}{\rightarrow} i \overset{\sigma}{\rightarrow} \sigma(i)$; while if $i = i_k$, then the action is $\sigma(i_k) \overset{\sigma^{-1}}{\rightarrow} i_k \overset{\gamma}{\rightarrow} i_{k+1} \overset{\sigma}{\rightarrow} \sigma(i_{k+1})$. However, this is exactly the effect of the cycle in our theorem. ■

Finally, as a prelude to the future optimization we are going to perform in §6.9 *Transpositional Canonicalization*, we remark that the subgroup formed from the transpositional symmetries is normal in the whole group $\mathcal{G}$. We can use our GAP example again to corroborate this.

```
gap> T:= Group(ST);
Group([ (1,2), (3,4), (5,6), (7,8), ( 9,10), (11,12) ])
gap> IsNormal(G,T);
true
gap> F:=FactorGroup(G,T);
<pc group with 5 generators>
gap> Size(F);
48
```

In fact let us prove this for any transpositional subgroup of a group.

Definition 6.8.G: The *transpositional* subgroup of a permutation group $\mathcal{G}$ is the group generated by the set of all transpositions in $\mathcal{G}$.

Theorem 6.8.D: The transpositional subgroup $\langle \mathcal{T} \rangle$ of a permutation group $\mathcal{G}$ is normal in $\mathcal{G}$, that is, $\langle \mathcal{T} \rangle \trianglelefteq \mathcal{G}$.

Proof: Let $\mathcal{H} = \langle \mathcal{T} \rangle$. First consider any conjugate of a transposition $\tau \in \mathcal{H}$, say $\tau^g = g \cdot \tau \cdot g^{-1}$ for $g \in \mathcal{G}$. By Theorem 6.8.C the cycle structure of the conjugated transposition must also be a transposition, and clearly τ^g is in $\mathcal{G}$, hence τ^g is in $\mathcal{H}$, by the definition of the transpositional subgroup. Thus $\tau^g \in \mathcal{H}$ for all $g \in \mathcal{G}$.

Now given any $\rho \in \mathcal{H}$, we know there must exist transpositions $\tau_1, \ldots, \tau_n \in \mathcal{H}$ such that $\rho = \tau_1 \cdot \tau_2 \cdot \ldots \cdot \tau_n$, again by the definition of a transpositional subgroup. Therefore, $g \cdot \rho \cdot g^{-1} = g \cdot \tau_1 \cdot \tau_2 \cdot \ldots \cdot \tau_n \cdot g^{-1} =$ $g \cdot \tau_1 \cdot g^{-1} \cdot g \cdot \tau_2 \cdot g^{-1} \cdot g \ldots \cdot g^{-1} \cdot g \cdot \tau_n \cdot g^{-1} = \tau_1^g \cdot \tau_2^g \cdot \ldots \cdot \tau_n^g$. But each of the conjugates of a transposition in $\mathcal{H}$ lies in $\mathcal{H}$, hence $g \cdot \rho \cdot g^{-1} \in \mathcal{H}$. Therefore $g \cdot \mathcal{H} \cdot g^{-1} = \mathcal{H}$, hence $\mathcal{H} \trianglelefteq \mathcal{G}$. ■

Later on, in section §6.9 *Transpositional Canonicalization*, we will show that we can essentially work entirely within the factor group $\mathcal{G}/\mathcal{T}$ when $\mathcal{T}$ can be generated by a set of adjacent transpositions. This greatly reduces the number of configurations that have to be considered in our canonicalization algorithm.

6.8.7 Testing of JRDES Generation

The forgoing subsections have introduced powerful theorems that will, amongst other things, guarantee the JRDES property for a generator set for a tensor product in terms of JRDES generator sets for the tensor factors. However, we are still left with the question of how to test if an arbitrary set of generators is JRDES? The answer, regrettably, is that sometimes one just has to perform this check by brute force. We could write a small piece of code in *Mathematica* for this purpose, but in practice we use this test infrequently. Therefore, let us use GAP to do this by brute force when the need arises. Here is a reasonably simple implementation of a function that tests if a given set of generators is JRDES.

```
testJRDES:= function(generators)
        local
orbit,orbits,G,maxOrbit,minOrbit,b,generatorsStabMin,generatorsStabMax;

        if generators = [] then return true; fi;

        G := Group(generators);
  orbits := Orbits(G);

        for orbit in orbits do
                maxOrbit:=Maximum(orbit);
```

```
        minOrbit:=Minimum(orbit);

        #check that we can move further right#
        for b in Filtered(orbit, p -> p < maxOrbit) do
                if Maximum(List(generators, s -> b^s)) <= b then
                        return false; fi; od;

        #check that we can move further left#
        for b in Filtered(orbit, p -> p > minOrbit) do
                if Minimum(List(generators, s -> b^s)) >= b then
                        return false; fi; od;

        #check that removing generators that move maxOrbit is the same as
         stabilizing on maxOrbit
        generatorsStabMax := Filtered(generators, s -> maxOrbit^s =
                                                        maxOrbit);
        if Group(Union( [()], generatorsStabMax)) <> Stabilizer(G,maxOrbit)
                then
                return false; fi;

        #check that removing generators that move minOrbit is the same as
         stabilizing on minOrbit
        generatorsStabMin := Filtered(generators, s -> minOrbit^s = minOrbit);
        if Group(Union( [()], generatorsStabMin)) <> Stabilizer(G,minOrbit) then
                return false; fi;

        #check to make sure the stabilized subgroups are JRDES
        if not( testJRDES(generatorsStabMax) and
                testJRDES(generatorsStabMin)) then
                return false; fi;

    od;

    return true;
end;
```

We can use this testing function to verify that the set of symmetries we have used for the Riemann tensor, namely { (1,2), (3,4), (1,3)(2,4) }, is indeed JRDES.

```
gap> testJRDES( [(1,2), (3,4), (1,3)(2,4)] );
true
```

In fact, by experimenting a little, one can find a set of non–adjacent transpositions which is not JRDES.

```
gap> testJRDES( [(1,2), (1,3), (1,4)] );
false
```

However, sometimes we can easily find an equivalent generating set that is JRDES.

```
gap> testJRDES( [(1,2), (2,3), (3,4)] );
true
gap> Group ( [(1,2), (1,3), (1,4)] ) = Group ( [(1,2), (2,3), (3,4)] );
true
```

Another common class of symmetries used in tensor analysis are the cyclical symmetries. These can be naturally JRDES generated.

```
gap> testJRDES( [(1,2,3), (1,3,2)] );
true
```

As an interesting non-trivial example, we can show that the overall set of tensor symmetries of the product of the three Riemannian tensors considered in §6.8.2 is JRDES.

```
gap> testJRDES( Union(ST,SX,SD) );
true
```

The last example above illustrates an important general result which is the topic of the next subsection.

6.8.8 Tensor Products and their JRDES Generators

This subsection contains the culmination of our development which guarantees that a signed permutation group can be "nicely" JRDES generated if its constituents can be "nicely" JRDES generated.

Theorem 6.8.E: A permutation group $\mathcal{G}$ corresponding to the symmetries of a tensor product of two versions of the tensor T can be "nicely" JRDES generated if the permutation group corresponding to T can be "nicely" JRDES generated.

Proof: Let us denote the permutation group corresponding to T by $\mathcal{H}$, which in turn is JRDES generated by $\mathcal{S}$. Then, by the definition of tensor products, $\mathcal{G} = \langle \mathcal{S} \bigcup \mathcal{S}' \bigcup \mathcal{S}_{\mathcal{D}} \rangle$ where $\mathcal{S}'$ is the shifted version of the generators $\mathcal{S}$ for the second tensor T and $\mathcal{S}_{\mathcal{D}}$ is the degeneracy symmetry of the overall tensor product. It is easy to show that the generators for $\mathcal{G}$ are directional since each point can either be moved internally by $\mathcal{H}$ or $\mathcal{H}'$, or flipped into its twin by the single degeneracy symmetry in $\mathcal{S}_{\mathcal{D}}$. Moreover, after we stabilize on any point, say β, the degeneracy symmetry involving this point is necessarily removed. Without loss of generality assume β occurs within the first tensor factor, that is $\mathcal{H}$. Then the stabilized subgroup, $\mathcal{G}_\beta$, is the internal direct product of $\mathcal{H}_\beta$ and $\mathcal{H}'$. Moreover, $\mathcal{H}_\beta$ and $\mathcal{H}'$ are JRDES generated by $\mathcal{S}_\beta$ and $\mathcal{S}'$, and the non-trivial orbits of $\mathcal{H}_\beta$ and $\mathcal{H}'$ obviously do not overlap. Therefore by Theorem 6.8.A, $\mathcal{G}_\beta$ is JRDES generated by $\mathcal{S}_\beta \bigcup \mathcal{S}'$. Hence, it

follows that $\mathcal{G}$ can be naturally JRDES generated by the generators of T along with the single degenerate name generator, that is, by $S \bigcup S' \bigcup S_{\mathcal{D}}$. ■

As before, the previous theorem can easily be generalized to a tensor product of n identical tensor factors.

Corollary 6.8.A: The permutation group of a tensor product can be naturally JRDES generated from the "nice" JRDES generator sets of the factors comprising the tensor product.

Proof: Obvious from Theorem 6.8.A and Theorem 6.8.E. ■

Previously, in §6.7.2 *Specification of Generators*, we saw how to enter and define new symmetries to be used in our canonicalization algorithm. In the current section we have shown how to obtain a JRDES generating set for a tensor product from the JRDES generating sets for the product factors. In §D.5 *Constructing Generators from Primitive Generators*, we capture the ideas of this section and develop the actual code for producing a generating set for a tensor product of factors in terms of the generator sets for the individual factors. Finally, the applications of the theory of this section are left to upcoming sections: adjacent transpositions play a key role in §6.9 *Transpositional Canonicalization*, while JRDES generators are used in §6.11.1 *Canonicalization of Free Indices*.

6.9 Transpositional Canonicalization

6.9.1 Transpositional Equivalence and Canonicalization

In this section we shall introduce machinery for the first of our two main optimizations which we make to the general algorithm. The second optimization, to be given in §6.11.1 *Canonicalization of Free Indices*, will put all the free indices in their canonical positions and then deal only with the stabilized subgroup of permutations that keep the free indices fixed.

The first optimization consists of treating as the same, all configurations that are related to each other by a sequence of allowed transpositions. That is, we partition the set of configurations into equivalence classes, where two configurations are in the same equivalence class if one can be reached from the other by the transpositions allowed by the symmetries. We then choose a representative from each such equivalence class. If we can still efficiently determine the answers we are looking for from the transpositional equivalence class representatives alone, we

will save a lot of work, since in general the number of equivalence classes of configurations is *much* smaller than the number of configurations. This means we have to do far less searching and far less generating of configurations, thus yielding a vast improvement in speed.

Formally, *transpositional equivalence* of two configurations $a, b \in C$ with respect to a set of transpositions $\mathcal{T}$ is a specific case of Definition 6.4.B given in §6.4.5 *Equivalence Classes*. That is, $a \sim_{\mathcal{T}} b$ if and only if $b = (\prod_i \tau_i) *_c a$ for some $\tau_i \in \mathcal{T}$. Thus, trivially, the $\mathcal{T}$-equivalence class containing a is just $\langle \mathcal{T} \rangle *_c a$.

To perform calculations with equivalence classes we of course need equivalence class representatives. Intuitively, let us use the smallest configuration in the equivalence class as the class representative. Formally, we choose the equivalence class representative for $\langle \mathcal{T} \rangle *_c a$ to be one element from $mins_c(\langle \mathcal{T} \rangle *_c a)$. Actually, we will see in §6.10 *Identically Zero Tensors* that $mins_c(\langle \mathcal{T} \rangle *_c a)$ contains a single element unless $-a \sim_{\mathcal{T}} a$. Thus, normally there is no choice at all and the class representative is the minimum configuration of the equivalence class.

Definition 6.9.A: For a set $\mathcal{T}$ of transpositions, the *transpositional canonicalization* operator, which we will denote by $\theta_{\mathcal{T}}$, takes a signed configuration a to the minimum signed configuration of the $\mathcal{T}$-equivalence class containing a (or one of the minimum signed configurations if there is more than one minimum). Formally, $\theta_{\mathcal{T}}(a) \in mins_c(\langle \mathcal{T} \rangle *_c a)$. Furthermore, we require $\theta_{\mathcal{T}}(a) = a$ if $a \in mins_c(\langle \mathcal{T} \rangle *_c a)$.

To reiterate, the underlying reason why we have to carefully state the definition of $\theta_{\mathcal{T}}$ is that our ordering operator, $\prec_c$, does not distinguish between signed configurations that differ only in sign. Consequently, a set of signed configurations might not have a unique minimum but rather two minimum signed configurations of the form $config_+$ and $config_-$. However, in the main, we can think of $\theta_{\mathcal{T}}(a)$ as the smallest signed configuration $\mathcal{T}$-equivalent to a.

So far we have not specified the value of $\theta_{\mathcal{T}}(a)$ in the case when $mins_c(\langle \mathcal{T} \rangle *_c a)$ contains two configurations. We will discuss this matter after the following result.

Theorem 6.9.A: $\theta_{\mathcal{T}}$ is a canonicalizing function (relative to $\sim_{\mathcal{T}}$) on signed configurations corresponding to non-zero tensors. Formally, for any signed configurations a and b which are not $\mathcal{T}$-zero-equivalent we have

(*i*) $\theta_{\mathcal{T}}(a) \sim_{\mathcal{T}} a$

(*ii*) $b \sim_{\mathcal{T}} a \;\Rightarrow\; \theta_{\mathcal{T}}(b) \equiv \theta_{\mathcal{T}}(a)$.

Proof: Trivially we have (*i*) by the definition of $\theta_{\mathcal{T}}$. For (*ii*) we have $b \sim_{\mathcal{T}} a$

$\Rightarrow b \sim_{\langle \mathcal{T} \rangle} a$

$\Rightarrow \langle \mathcal{T} \rangle *_c b = \langle \mathcal{T} \rangle *_c a$ by Lemma 6.7.A

$\Rightarrow mins_c(\langle \mathcal{T} \rangle *_c b) = mins_c(\langle \mathcal{T} \rangle *_c a)$

$\Rightarrow \theta_{\mathcal{T}}(b) = \theta_{\mathcal{T}}(a)$ by definition of $\theta_{\mathcal{T}}$, since $mins_c(\langle \mathcal{T} \rangle *_c a)$ contains just one minimum.

$\Rightarrow \theta_{\mathcal{T}}(b) \equiv \theta_{\mathcal{T}}(a)$ since equal signed configurations must be identical. ■

It is clear that Definition 6.9.A does not completely specify $\theta_{\mathcal{T}}(a)$ when a is in some $\mathcal{T}$-equivalence class which has two minimum configurations. One convention for the choice of

$\theta_{\mathcal{T}}(a)$, when there is no unique minimum, would be the minimum signed configuration having the same sign as the configuration a. Unfortunately, this convention does not respect property (*ii*) of a canonicalizing function (Definition 6.7.A), namely, it would no longer always be true that $b \sim_{\mathcal{T}} a \Rightarrow \theta_{\mathcal{T}}(b) \equiv \theta_{\mathcal{T}}(a)$. For example, if $\mathcal{T}$ is such that some $\mathcal{T}$-equivalence class has two minimum configurations, say c and $-c$, then it would follow that $-c \sim_{\mathcal{T}} c$ yet $\theta_{\mathcal{T}}(-c) = -c = -\theta_{\mathcal{T}}(c)$.

Besides the above mentioned candidate, there are obviously many more possibilities for $\theta_{\mathcal{T}}$. At the theoretical level we will not make any specific choice for $\theta_{\mathcal{T}}$. Instead we will just let $\theta_{\mathcal{T}}$ denote a definite but unspecified one of the many such candidates. However, our implementation in *Mathematica* of $\theta_{\mathcal{T}}$ will correspond to a completely specified computable function, to be called $\Theta_{\mathcal{T}}$.

In general the function $\theta_{\mathcal{T}}$will not be a canonicalizing function on C, the set of all signed configurations. However, it is extremely close to being a canonicalizing function but for the uncertainty in the sign of the result.

Theorem 6.9.B: $\theta_{\mathcal{T}}$is a *semi-canonicalizing* function (relative to $\sim_{\mathcal{T}}$) on the set of signed configurations. That is, for any signed configurations a and b we have

(*i*) $\theta_{\mathcal{T}}(a) \sim_{\mathcal{T}} a$ and

(*ii*) $b \sim_{\mathcal{T}} a \Rightarrow \theta_{\mathcal{T}}(b) \equiv \pm\theta_{\mathcal{T}}(a)$,

Technical Note: $\theta_{\mathcal{T}}(b) \equiv \pm\theta_{\mathcal{T}}(a)$ means that either $\theta_{\mathcal{T}}(b) \equiv \theta_{\mathcal{T}}(a)$ or $\theta_{\mathcal{T}}(b) \equiv -\theta_{\mathcal{T}}(a)$.

Proof: (*i*) is trivial, as before. The proof of (*ii*) is just a trivial modification of the proof of Theorem 6.9.A (*ii*). If $\text{mins}_c(\langle\mathcal{T}\rangle *_c a)$ contains two elements, say c and $-c$, then $\theta_{\mathcal{T}}(b)$ equals either c or $-c$, and similarly for $\theta_{\mathcal{T}}(a)$. Therefore $\theta_{\mathcal{T}}(b) \equiv \pm\theta_{\mathcal{T}}(a)$. ■

It is convenient for us to overload our definition of $\theta_{\mathcal{T}}$ so that it applies not only to signed configurations but also to sets of signed configurations. $\theta_{\mathcal{T}}$ is overloaded in the obvious way, that is, for any set of signed configurations $\mathcal{A}$, $\theta_{\mathcal{T}}(\mathcal{A}) = \{\theta_{\mathcal{T}}(a) \mid a \in \mathcal{A}\}$. We will define $\Theta_{\mathcal{T}}[\ell st]$ similarly.

The question now arises as to how to calculate $\theta_{\mathcal{T}}(a)$ given a signed configuration a. Following our definition of $\theta_{\mathcal{T}}$, one could generate all the configurations transpositionally equivalent to a, that is $\langle\mathcal{T}\rangle *_c a$, using `generateConfigurations[`$a$`,` $\mathcal{T}$`]`, and then pick a minimum element from this set. Obviously though, this would mean we have saved nothing and generated all the configurations in the end anyway. What is needed is a fast way to determine $\theta_{\mathcal{T}}(a)$ given a. Luckily, such a method exists. For the remainder of this section we shall develop a method for efficiently obtaining a minimum equivalence class representative. It is this method that will be embodied in our implementation of the *Mathematica* function $\Theta_{\mathcal{T}}$, which is an instantiation of the theoretical function $\theta_{\mathcal{T}}$.

By using the upcoming method, we can efficiently calculate a minimum representative for the $\mathcal{T}$-equivalence class containing a, that is, we can efficiently calculate $\theta_{\mathcal{T}}(a)$. However, to find the overall canonical configuration, we need at least one such minimum representative from each $\mathcal{T}$-equivalence class contained in $\mathcal{G} *_c t$, that is, we essentially need to calculate $\theta_{\mathcal{T}}(\mathcal{G} *_c t)$. In §6.9.4 *Transpositional Canonicalization Algorithm*, we formally present the algorithm for making such a

calculation, an algorithm which avoids first generating $\mathcal{G} *_c t$. Finally, in §6.9.6 *Correctness Proof for GenerateConfigurations*$_{\mathcal{T}}$, we will prove that this algorithm actually does produce a set containing at least one minimum signed configuration from each of the $\mathcal{T}$-equivalence classes whose members make up $\mathcal{G} *_c t$.

6.9.2 ReducingSwapQ

In working towards our goal for an efficient implementation of $\theta_{\mathcal{T}}$, let us define a function, `reducingSwapQ`, which tells us if we should apply a signed transposition to a signed configuration. In particular, given $a \in C$ and a transposition $(i \leftrightarrow j)$ with $i < j$, then `reducingSwapQ` determines whether $(i \leftrightarrow j)_{\pm} *_c a \prec_c a$ is true or not. Moreover, `reducingSwapQ` will accomplish this without actually performing the transposition and relabelling on the configuration a. `reducingSwapQ` takes two indices, say a_i and a_j with $i < j$, of the configuration a, and from these indices determines if transposing them will result in a "smaller" configuration. Motivated by the fact that applying the transposition $(i \leftrightarrow j)$ to the configuration a interchanges a_i and a_j, we will suggestively denote the test `reducingSwapQ[`a_i, a_j`]` by $a_i \leftrightarrows_? a_j$.

```
In[1]:= Notation[a_ ⇆? b_ ⇔ reducingSwapQ[a_, b_]]
```

We will intentionally implement the `reducingSwapQ` test in a non-symmetrical way. First, recall that each admissible index of a configuration contains information specifying at which position it occurs, since the i^{th} index in any configuration will be of the form `s[`i,j`]`, `High[`*name*,i`]` or `Low[`*name*,i`]`. For convenience, we will only implement $a \leftrightarrows_? b$ in the case when the inherent position of a is to the left of the inherent position of b. This requirement that a appears to the left of b makes our definition for `reducingSwapQ` easier to state, allows faster execution, and conforms with all of the algorithms that use `reducingSwapQ` since they always pass index arguments to `reducingSwapQ` in this order. It should also be noted that though the test $a \leftrightarrows_? b$ is used when the indices a and b are taken from some configuration, nevertheless the definition of $a \leftrightarrows_? b$, which we next give, does not refer to any specific configuration.

```
In[2]:= _s ⇆? _High = True;
        _High ⇆? _s = False;
        _s ⇆? _Low = False;
        _Low ⇆? _s = True;
        _High ⇆? _Low = False;
        _Low ⇆? _High = True;
        b_↑ ⇆? a_↑ := a ≺lex b;
        b_↓ ⇆? a_↓ := a ≺lex b;
        s_i_,m_ ⇆? s_j_,n_ =
           i < j < n < m ∨ n < m < i < j ∨ Max[i, m] < Min[j, n];
```

The only unintuitive part in our definition is for $s_{i,m} \leftrightarrows_? s_{j,n}$, and in this case what we specify will be exactly what we need later to prove Theorem 6.9.C. Actually, in the interests of speed, let us also compile this case of the definition of $a \leftrightarrows_? b$.

```
In[11]:= compareS = Compile[
             {{i, _Integer}, {j, _Integer}, {m, _Integer}, {n, _Integer}},
             i < j < n < m ∨ n < m < i < j ∨ Max[i, m] < Min[j, n]];

In[12]:= s_{i_,m_} ⇆? s_{j_,n_} := compareS[i, j, m, n];
```

Here are some simple examples involving the reducing swap test operator.

```
In[13]:= {s_{1,4} ⇆? s_{2,3}, s_{3,2} ⇆? s_{4,1}, f_1^↓ ⇆? s_{2,3}, s_{2,3} ⇆? f_3^↓}

Out[13]= {True, True, True, False}
```

Our definition of $\leftrightarrows?$ gives incorrect answers when the arguments are entered in the incorrect order. For example, we know that transposing $\mathtt{f}_3^{\uparrow}$ and $\mathtt{s}_{1,4}$ would result in a smaller configuration since it moves a free high index from position 3 to position 1 while moving a summed index from position 1 to position 3. However, we get

```
In[14]:= f_3^↑ ⇆? s_{1,4}

Out[14]= False
```

6.9.3 Proof that Reducing Transpositions Yield Smaller Configurations

We now state and prove the theorem which will underpin the transpositional canonicalization algorithm we are seeking to define. The following theorem simply proves that if the reducing transposition test is true for the given indices of a configuration, then applying the transposition to the indices will yield a smaller or reduced configuration.

Theorem 6.9.C: Given any signed configuration $a \in C$ and a signed transposition $(i \leftrightarrow j)_{\pm}$ such that $i < j \leqslant a_{ten}$ then $a_i \leftrightarrows? a_j$ if and only if $(i \leftrightarrow j)_{\pm} *_c a \prec_c a$.

Proof: For convenience in this proof, we ignore all signs and consider just configurations and transpositions. The configuration $(i \leftrightarrow j) *_c a$ only differs from the configuration a on the indices a_i and a_j, and possibly their connected indices, $a_{\bar{i}}$ and $a_{\bar{j}}$. Thus, the relative ordering of $(i \leftrightarrow j) *_c a$ and a will be determined solely on these indices where the two configurations can differ. Also, the truth value of $a_i \leftrightarrows? a_j$ is of course determined solely by a_i and a_j. Consequently, we can ignore the rest of the indices for the duration of this proof.

There are various cases to consider, depending upon whether the indices a_i and a_j are high, low or summed. If the a_i and a_j indices are of different types, then by inspection of the definition of our ordering, $\prec_c$, and the definition of `reducingSwapQ`, it is relatively easy to see that our theorem is true. For instance, if a_i is a summed index and a_j is a free high index, then clearly we will achieve a smaller configuration by transposing i and j, since a free index will be further to the left while a summed index will be further to the right. Also, it is clear in this instance that $a_i \leftrightarrows_? a_j$ is true.

Next, consider the case when a_i and a_j are both high or both low. Then it is easy to see that the ordering of $(i \leftrightarrow j) *_c a$ relative to a is determined solely on the lexical order of the indices a_i and a_j. Similarly, for this case, $a_i \leftrightarrows_? a_j$ is also determined by the lexical order of a_i and a_j. By inspection of the definition of our ordering and the definition of `reducingSwapQ`, it is clear that in this case the theorem is true.

Therefore, it remains to show that if $a_i = \mathtt{s_{i,m}}$ and $a_j = \mathtt{s_{j,n}}$, then $\mathtt{s_{i,m}} \leftrightarrows_? \mathtt{s_{j,n}}$ is true if and only if the transposed configuration a', where the i^{th}, j^{th}, m^{th} and n^{th} indices have become $\mathtt{s_{i,n}}$, $\mathtt{s_{j,m}}$, $\mathtt{s_{m,j}}$ and $\mathtt{s_{n,i}}$, is smaller that the configuration a, where the i^{th}, j^{th}, m^{th} and n^{th} indices are $\mathtt{s_{i,m}}$, $\mathtt{s_{j,n}}$, $\mathtt{s_{m,i}}$ and $\mathtt{s_{n,j}}$. The case when $m = j$ and $n = i$, that is $a_i = \mathtt{s_{i,j}}$ and $a_j = \mathtt{s_{j,i}}$, is trivial. Thus we need only consider the case when i, j, m, n are all distinct.

As stated above, let us temporarily ignore the other indices besides i, j, m and n, which are the same for both configurations. We will refer to these configurations, where we have ignored indices that are the same, as *mini-configurations*. For instance, the mini-configuration $\{\mathtt{s_{i,m}}, \mathtt{s_{j,n}}, \mathtt{s_{n,j}}, \mathtt{s_{m,i}}\}$ corresponds to a full configuration of the form $\{\ldots, \mathtt{s_{i,m}}, \ldots, \mathtt{s_{j,n}}, \ldots, \mathtt{s_{n,j}}, \ldots, \mathtt{s_{m,i}}, \ldots\}_{\pm}$. We will deduce information about how mini-configurations compare to each other, and from this we can infer information on the orderings of the full configurations. Our goal is to find all mini-configurations such that transposing and relabeling positions i and j results in a smaller configuration. From this we can obtain a criterion that i, j, m and n must satisfy. Finally, we must show that this criterion is simply the one we defined for $\mathtt{s_{i,m}} \leftrightarrows_? \mathtt{s_{j,n}}$.

We must consider all 24 different reorderings of $\{\mathtt{s_{i,m}}, \mathtt{s_{j,n}}, \mathtt{s_{m,i}}, \mathtt{s_{n,j}}\}$ and determine which mini-configurations would be made smaller by a transposition of i and j. Unfortunately, our existing routine to compare configurations does not work for symbolic values. However, it is easily seen that the ordering of one configuration over another is only dependent on the relative ordering of i, j, m and n, so we can compare mini-configurations by simply substituting any numerical values that maintain the relative ordering of the symbolic values. The following function substitutes such relative values.

```
In[15]:= relativeValues [expr : {s_{w_,_}, s_{x_,_}, s_{y_,_}, s_{z_,_}}] :=
           expr /. {w → 1, x → 2, y → 3, z → 4}
```

Transposing and relabeling positions i and j can be symbolically achieved by using the following replacements: $\mathtt{s_{i,m}} \to \mathtt{s_{i,n}}$, $\mathtt{s_{m,i}} \to \mathtt{s_{m,j}}$, $\mathtt{s_{j,n}} \to \mathtt{s_{j,m}}$, $\mathtt{s_{n,j}} \to \mathtt{s_{n,i}}$.

```
In[16]:= transposeIandJ @ expr_ :=
           expr /. {s_{i,m} → s_{i,n}, s_{m,i} → s_{m,j}, s_{j,n} → s_{j,m}, s_{n,j} → s_{n,i}}
```

We can now determine if transposing and relabeling positions i and j in a mini–configuration will result in a smaller mini–configuration.

```
In[17]:= transposedConfigurationIsSmallerQ @ reducedConfiguration_ :=
           Block[{transposedConfiguration = transposeIandJ @ reducedConfiguration},
             relativeValues[transposedConfiguration]+ <c
               relativeValues[reducedConfiguration]+]
```

For instance, if we transpose i and j in the mini–configuration $\{s_{i,m}, s_{j,n}, s_{n,j}, s_{m,i}\}$, we get the mini–configuration $\{s_{i,n}, s_{j,m}, s_{n,i}, s_{m,j}\}$.

```
In[18]:= transposeIandJ @ {s_{i,m}, s_{j,n}, s_{n,j}, s_{m,i}}

Out[18]= {s_{i,n}, s_{j,m}, s_{n,i}, s_{m,j}}
```

This mini–configuration is smaller than the untransposed configuration.

```
In[19]:= transposedConfigurationIsSmallerQ @ {s_{i,m}, s_{j,n}, s_{n,j}, s_{m,i}}

Out[19]= True
```

It is now easy to obtain all mini–configurations that are made smaller by transposing i and j where $i < j$.

```
In[20]:= miniConfigurations = Select[Permutations[{s_{i,m}, s_{j,n}, s_{m,i}, s_{n,j}}],
           transposedConfigurationIsSmallerQ[miniConfig] ∧
             Position[miniConfig, s_{i,m}][[1, 1]] <
               Position[miniConfig, s_{j,n}][[1, 1]] &miniConfig]

Out[20]= {{s_{i,m}, s_{j,n}, s_{n,j}, s_{m,i}}, {s_{i,m}, s_{m,i}, s_{j,n}, s_{n,j}}, {s_{i,m}, s_{m,i}, s_{n,j}, s_{j,n}},
           {s_{m,i}, s_{i,m}, s_{j,n}, s_{n,j}}, {s_{m,i}, s_{i,m}, s_{n,j}, s_{j,n}}, {s_{n,j}, s_{m,i}, s_{i,m}, s_{j,n}}}
```

From each mini–configuration that can be made smaller by a transposition on i and j, we obtain a condition on i, j, m and n.

```
In[21]:= relativeOrdering@ {s_{w_,_}, s_{x_,_}, s_{y_,_}, s_{z_,_}} := w < x < y < z

In[22]:= criterion = Or @@ relativeOrdering /@ miniConfigurations

Out[22]= i < j < n < m || i < m < j < n || i < m < n < j ||
           m < i < j < n || m < i < n < j || n < m < i < j
```

It is clear that the 2nd, 3rd, 4th and 5th disjuncts above are in this instance collectively equivalent to Max[i, m] < Min[j, n]. Thus it is obvious that the above criterion is equivalent to the definition given for $s_{i,m} \leftrightarrows_? s_{j,n}$, namely

$$i < j < n < m \;\vee\; n < m < i < j \;\vee\; \text{Max}[i, m] < \text{Min}[j, n] \tag{6.9.a}$$

Since the mini-configuration is made smaller by the transposition ($i \leftrightarrow j$) whenever $s_{i,m} \leftrightarrows_? s_{j,n}$ is true (and vice versa), it follows that the configuration will be made

smaller by the transposition $(i \leftrightarrow j)$ whenever $s_{i,m} \leftrightarrows_? s_{j,n}$ is true (and vice versa).
■

For the interested reader, here is an alternative to the above proof. Although this alternative proof furnishes no insight into how the test of reducingSwapQ was ever created, it proves the theorem with a minimum of effort.

Alternative Proof: As before, all orderings can be determine from mini-configurations. If we show that the result is true for all types of mini-configurations, then we have shown the theorem. So by brute force we will construct the set of all possible labeled mini–configurations. We start this by constructing all possible subsets of length 4 on the set $\{a^+, b^+, c^+, d^+, a^-, b^-, c^-, d^-\}$.

```
In[23]:= allConfigurations = Flatten[
           Permutations /@ KSubsets[{a⁺, b⁺, c⁺, d⁺, a⁻, b⁻, c⁻, d⁻}, 4], 1];
```

To give the reader an idea of what some of these possible configurations look like, let us consider 6 configurations chosen at random from the set of all configurations.

```
In[24]:= RandomKSubset[allConfigurations, 6]

Out[24]= {{c⁺, b⁺, a⁺, a⁻}, {c⁻, d⁺, b⁺, a⁺}, {c⁺, b⁻, a⁻, a⁺},
           {c⁻, a⁻, d⁺, a⁺}, {a⁻, d⁻, d⁺, b⁺}, {a⁻, d⁻, c⁺, b⁻}}
```

Let us now convert these configurations to signed configurations that are standardly labeled.

```
In[25]:= allSignedConfigurations =
           Union[toSignedConfiguration /@ (Tensor[T, #] & /@ allConfigurations)];
```

The set *allSignedConfigurations* now contains all possible labeled signed mini–configurations. Let us again get an idea of what some of these signed configurations look like by examining 6 signed configurations, chosen at random from the set of all signed configurations.

```
In[26]:= RandomKSubset[allSignedConfigurations, 6]

Out[26]= {{a₁↑, d₂↓, b₃↓, c₄↓}₊, {b₁↑, d₂↓, a₃↓, c₄↓}₊, {b₁↑, s₂,₄, c₃↓, s₄,₂}₊,
           {d₁↑, a₂↑, c₃↓, b₄↓}₊, {d₁↑, b₂↓, c₃↓, a₄↓}₊, {s₁,₃, a₂↓, s₃,₁, c₄↑}₊}
```

Now all that remains is to show that our reducingSwapQ operator yields true if and only if the signed configuration is made smaller by the transposition. To show this, let us introduce a function that takes both a signed transposition and a signed configuration, and returns true if reducingSwapQ accurately predicts whether performing the transposition yields a lower configuration.

```
In[27]:= testHypothesis[sτ : (i_ ↔ j_)_signτ_, sc_signedConfiguration] :=
           (sτ *_c sc <_c sc) == (sc[[2,i]] ⇆? sc[[2,j]])
```

We can, for instance, test this hypothesis with the transposition $(1 \leftrightarrow 3)_+$ and the configuration $\{s_{1,2}, s_{2,1}, a_3^\uparrow, d_4^\uparrow\}_+$.

```
In[28]:= testHypothesis[(1 ↔ 3)+, {s1,2, s2,1, a3↑, d4↑}+]

Out[28]= True
```

So now all we do is test the hypothesis on every transposition on 1 to 4 with every configuration.

```
In[29]:= Union @ Flatten @ Outer[testHypothesis, {(1 ↔ 2)+, (1 ↔ 3)+, (1 ↔ 4)+,
                  (2 ↔ 3)+, (2 ↔ 4)+, (3 ↔ 4)+}, allSignedConfigurations]

Out[29]= {True}
```

Thus by exhaustively testing all possible transpositions with all possible mini-configurations, we have shown the following: `reducingSwapQ` returns `True` on a mini–configuration, hence on the full configuration, if and only if the transposition makes the configuration smaller. ■

6.9.4 Transpositional Canonicalization and GenerateConfigurations$_{\mathcal{T}}$

Now that we know that we can get smaller configurations through using the reducing swap operator, we can implement the transpositional canonicalization operator, $\theta_{\mathcal{T}}$.

On the basis of extremely strong evidence, given in §C.2 *Evidence for Steepest Descent Conjecture*, the optimized algorithm is predicated on the following conjecture. Given a set of adjacent transpositions $\mathcal{T}$, when we successively apply transpositions that yield smaller and smaller configurations until we cannot obtain any smaller configurations through these transpositions, then we obtain the smallest $\mathcal{T}$-equivalent configuration possible. In essence, successively transforming in a "downhill" manner will yield a local minimum, and this minimum will in fact be a global minimum over the $\mathcal{T}$-equivalence class.

Conjecture 6.9.A: Given a signed configuration $a \in C$ and a set of adjacent transpositions $\mathcal{T}$ such that $\forall\, \tau \in \mathcal{T},\ a \preccurlyeq_c \tau *_c a$, then $\theta_{\mathcal{T}}(a) = a$.

Evidence: See the arguments in §C.2 *Evidence for Steepest Descent Conjecture*. ■

Corollary 6.9.A: Given a signed configuration $a \in C$ and a set of adjacent transpositions $\mathcal{T}$, then by applying members of $\mathcal{T}$ in a "downhill" manner, one obtains a minimum of the transpositional equivalent configurations. Formally, any maximal sequence of transpositions $\tau_i \in \mathcal{T}$ which yield successively smaller configurations a_i will terminate. The final such configuration satisfies $a_{final} \in mins_c(\langle \mathcal{T} \rangle *_c a)$.

Let us proceed to incorporate Conjecture 6.9.A into our algorithms, and then subsequently examine the justification / evidence for it once we can compare results obtained using the conjecture to results obtained without using the conjecture. In §C.2.15 *Limits to Testing the*

Conjecture we will exhaustively verify this conjecture up to order 10 on all possible configurations with all possible transpositional orbits. This will involve around 4.5 million tests of the conjecture, thus giving a strong indication that it is true.

```
In[30]:= Notation[Θ_𝒯_[c_] ⟺ transpositionallyCanonicalize[c_, 𝒯_]]
```

In our final algorithm, we will only include adjacent transpositions in our set of transpositional generators. However, we frame our definition of $\Theta_{\mathcal{T}}(c)$ in terms of arbitrary transpositions. One repeatedly applies each $\mathcal{T}$-transposition in succession if it lowers the configuration, until there is no change. We keep a local copy of the signed configuration so that we can destructively update it.

```
In[31]:= Θ_𝒯_List[signedConfigs_List] := Θ_𝒯[sc] &_sc /@ signedConfigs;
         Θ_𝒯_List[configuration_signC_] :=
           Block[
             {newConfiguration = configuration, newSign = signC, partialCanonicalConfiguration = {}},
               While[partialCanonicalConfiguration ≠ newConfiguration,
                     partialCanonicalConfiguration = newConfiguration;
                     transposeIfMoreCanonical /@ 𝒯];
             partialCanonicalConfiguration_newSign]

In[33]:= transposeIfMoreCanonical @ (i_ ↔ j_)_signτ_ :=
           If[newConfiguration[[i]] ↮? newConfiguration[[j]], swapThem[signτ, i, j]]

In[34]:= swapThem[signτ_, i_, j_] :=
           ({newConfiguration[[i]], newConfiguration[[j]]} =
                 {newConfiguration[[j]], newConfiguration[[i]]};
            newSign = newSign signτ;
            newConfiguration = Replace[newConfiguration, {i → j, j → i}, {2}]);
```

It should be clear from Theorem 6.9.C and Conjecture 6.9.A, at least in the case when $\mathcal{T}$ consists of just adjacent transpositions, that $\Theta_{\mathcal{T}}$ is an implementation of the transpositional canonicalization operator, $\theta_{\mathcal{T}}$.

Let us demonstrate transpositional canonicalization on a tensor product of three Riemannian tensors.

```
In[35]:= signedConfig = ⟨R^{c a m n} R_a^{b d} R_{c n m b d}⟩_c

Out[35]= {S_{1,8}, S_{2,5}, S_{3,10}, S_{4,9}, S_{5,2}, S_{6,11}, S_{7,12}, S_{8,1}, S_{9,4}, S_{10,3}, S_{11,6}, S_{12,7}}_+

In[36]:= {S_𝒯, S_𝒳, S_𝒟} = ⟨R^{a b c d} R^{m n} R_{b a} R_{d c n m}⟩_{S_{𝒯,𝒳,𝒟}}

Out[36]= {{(1 ↔ 2)_, (3 ↔ 4)_, (5 ↔ 6)_, (7 ↔ 8)_, (9 ↔ 10)_, (11 ↔ 12)_},
           {{3, 4, 1, 2, 5, 6, 7, 8, 9, 10, 11, 12}_+,
              {1, 2, 3, 4, 7, 8, 5, 6, 9, 10, 11, 12}_+,
              {1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 9, 10}_+},
           {{5, 6, 7, 8, 1, 2, 3, 4, 9, 10, 11, 12}_+,
              {1, 2, 3, 4, 9, 10, 11, 12, 5, 6, 7, 8}_+}}

In[37]:= Θ_S𝒯[signedConfig]
```

```
Out[37]= {s1,6, s2,8, s3,9, s4,10, s5,11, s6,1, s7,12, s8,2, s9,3, s10,4, s11,5, s12,7}_

In[38]:= Union @ Θ_{S_T} [generateConfigurations [signedConfig, S_T ⋃ S_X ⋃ S_D]]

Out[38]= {{s1,5, s2,6, s3,9, s4,11, s5,1, s6,2, s7,10, s8,12, s9,3, s10,7, s11,4, s12,8}_,
           {s1,5, s2,9, s3,6, s4,10, s5,1, s6,3, s7,11, s8,12, s9,2, s10,4, s11,7, s12,8}_,
           {s1,5, s2,11, s3,6, s4,12, s5,1, s6,3, s7,9, s8,10, s9,7, s10,8, s11,2, s12,4}_,
           {s1,6, s2,8, s3,9, s4,10, s5,11, s6,1, s7,12, s8,2, s9,3, s10,4, s11,5, s12,7}_,
           {s1,6, s2,8, s3,11, s4,12, s5,9, s6,1, s7,10, s8,2, s9,5, s10,7, s11,3, s12,4}_,
           {s1,7, s2,8, s3,9, s4,11, s5,10, s6,12, s7,1, s8,2, s9,3, s10,5, s11,4, s12,6}_,
           {s1,7, s2,9, s3,8, s4,10, s5,11, s6,12, s7,1, s8,3, s9,2, s10,4, s11,5, s12,6}_,
           {s1,7, s2,11, s3,8, s4,12, s5,9, s6,10, s7,1, s8,3, s9,5, s10,6, s11,2, s12,4}_,
           {s1,9, s2,10, s3,6, s4,8, s5,11, s6,3, s7,12, s8,4, s9,1, s10,2, s11,5, s12,7}_,
           {s1,9, s2,11, s3,5, s4,6, s5,3, s6,4, s7,10, s8,12, s9,1, s10,7, s11,2, s12,8}_,
           {s1,9, s2,11, s3,7, s4,8, s5,10, s6,12, s7,3, s8,4, s9,1, s10,5, s11,2, s12,6}_,
           {s1,11, s2,12, s3,6, s4,8, s5,9, s6,3, s7,10, s8,4, s9,5, s10,7, s11,1, s12,2}_}
```

Given that we can now efficiently transpositionally canonicalize a configuration, we can update our algorithm that generates all possible configurations. This new version of `generateConfigurations` is almost exactly the same as the previous version except we transpositionally canonicalize all new configurations before we add them to the collection of found configurations. In the following, as before, $\mathcal{F}$ is the collection of configurations found in previous stages and $\mathcal{N}$ is the collection of new configurations found at the current stage. However, unlike before, instead of using all of $\mathcal{S}$ as our "primary generating set" we now use only the non-transpositional generators, that is $\mathcal{R} = \mathcal{S} \setminus \mathcal{T}$.

```
In[39]:= Notation[generateConfigurations_{T_} [sc_, R_] ⟺
           generateTranspositionallyCanonicalConfigurations [sc_, R_, T_]]

In[40]:= generateConfigurations_{T_List} [
           seedConfiguration_signedConfiguration, R_List] :=
         Block[{F = {}, N = {Θ_T [seedConfiguration]}},
           While [N ≠ {},
             F = F ⋃ N;
             N = Θ_T [R *_c N] \ F;];
           F]
```

It should be noted that the new refinement, $\texttt{generateConfigurations}_{\mathcal{T}}$, has some significant differences from the old `generateConfigurations`. The new version has three arguments whereas the old version just had two. Functionally, the two algorithms differ significantly in what is added to $\mathcal{F}$ at any particular stage. In the old `generateConfigurations`, any newly generated signed configuration was added to $\mathcal{F}$. In the new $\texttt{generateConfigurations}_{\tau}$ only transpositionally canonicalized signed configurations are added, with each such configuration being a minimum element in the $\mathcal{T}$-equivalence class of some intermediately generated element.

6.9.5 Comparisons

For the purposes of illustration, let us perform a comparison on the size of the sets produced by `generateConfigurations` and $\texttt{generateConfigurations}_{\tau}$.

```
In[41]:= seedConfig = ⟨R^{camn} R_{a c}^{b d} R_{nmbd}⟩_c ;
```

As in the previous subsection, we obtain the symmetries as follows.

```
In[42]:= {S_T, S_X, S_D} = ⟨R^{camn} R_{a c}^{b d} R_{nmbd}⟩_{S_T,X,D} ;
```

We can now find all equivalent signed configurations for our tensor product under its symmetries.

```
In[43]:= generateConfigurations[seedConfig, S_T ⋃ S_X ⋃ S_D] // Length // Timing
Out[43]= {2.33333 Second, 384}
```

Thus we can see that there are 384 different configurations equivalent to the initial configuration. The number of configurations is starting to make the computation time of the algorithm approach an undesirable amount. By removing some of the redundant generators, we can improve the generation time.

```
In[44]:= generateConfigurations[seedConfig, S_T⟦{1, 2}⟧ ⋃ S_X⟦{1}⟧ ⋃ S_D] // Length //
           Timing
Out[44]= {1.15 Second, 384}
```

However, there are still a large number of configurations generated. This collection can be greatly narrowed by using transpositional canonicalization.

```
In[45]:= generateConfigurations_{S_T}[seedConfig, S_X ⋃ S_D] // Length // Timing
Out[45]= {0.166667 Second, 12}
```

Moreover, we can further reduce this computation time by eliminating redundant generators.

```
In[46]:= generateConfigurations_{S_T}[seedConfig, S_X⟦{1}⟧ ⋃ S_D] // Length // Timing
Out[46]= {0.116667 Second, 12}
```

Actually, we will be able to do even better than this with a small amount of temporary caching.

For a correctness check, it is nice to verify that the configurations returned by `generateConfigurations` and $\texttt{generateConfigurations}_{\mathcal{T}}$ agree. We can confirm this by taking the union of the transpositionally canonicalized results of `generateConfigurations` and comparing them to the results of $\texttt{generateConfigurations}_{\mathcal{T}}$.

```
In[47]:= Union[Θ_{S_T}[generateConfigurations[seedConfig, S_T ⋃ S_X ⋃ S_D]]] ≡
           generateConfigurations_{S_T}[seedConfig, S_X ⋃ S_D]

Out[47]= True
```

6.9.6 Correctness Proof for GenerateConfigurations$_{\mathcal{T}}$

We now must prove that given an initial configuration $t \in C$ corresponding to a tensor product which has the permutation group symmetries of $\mathcal{G}$, our revised algorithm actually generates all, or at least a sufficient subset, of $\theta_{\mathcal{T}}(\mathcal{G} *_c t)$. Let us first prove an intermediate lemma.

Lemma 6.9.A: Assume that the transpositional subgroup of $\mathcal{G} = \langle \mathcal{S} \rangle$ can be generated by the set of transpositions $\mathcal{T}$. Then for any $\rho \in \mathcal{S}$ and signed configuration $a \in C$, $\theta_{\mathcal{T}}(\rho *_c a) = \pm\theta_{\mathcal{T}}(\rho *_c \theta_{\mathcal{T}}(a))$.

Proof: First observe that $\rho \cdot \tau = \rho \cdot \tau \cdot \rho^{-1} \cdot \rho = \tau^{\rho} \cdot \rho$ for any $\rho \in \mathcal{G}$ and $\tau \in \mathcal{T}$. Now, by definition, $\theta_{\mathcal{T}}(a) \sim_{\mathcal{T}} a$, hence $\theta_{\mathcal{T}}(a) = \tau_1 \cdot \tau_2 \cdot \ldots \cdot \tau_n *_c a$ for some $\tau_i \in \mathcal{T}$. Therefore $\rho *_c \theta_{\mathcal{T}}(a) = \rho *_c (\tau_1 \cdot \tau_2 \cdot \ldots \cdot \tau_n *_c a) = (\rho \cdot \tau_1 \cdot \tau_2 \cdot \ldots \cdot \tau_n) *_c a$, and hence by our observation we can commute the τ_i through the ρ to yield $\rho *_c \theta_{\mathcal{T}}(a) = (\tau_1^{\rho} \cdot \tau_2^{\rho} \cdot \ldots \cdot \tau_n^{\rho} \cdot \rho) *_c a$. By Theorem 6.8.C each τ_i^{ρ} must be a transposition since each τ_i is a transposition, thus each τ_i^{ρ} must be in the transpositional subgroup $\langle \mathcal{T} \rangle$. Therefore $\rho *_c \theta_{\mathcal{T}}(a) = (\tau_1^{\rho} \cdot \tau_2^{\rho} \cdot \ldots \cdot \tau_n^{\rho}) *_c (\rho *_c a) \sim_{\mathcal{T}} \rho *_c a$. Consequently, by Theorem 6.9.A which shows that $\theta_{\mathcal{T}}$ is a semi-canonicalizing function, it follows that $\theta_{\mathcal{T}}(\rho *_c a) = \pm\theta_{\mathcal{T}}(\rho *_c \theta_{\mathcal{T}}(a))$. ■

The main result is now easy to show. By construction, $\Theta_{\mathcal{T}}(\mathcal{G} *_c t)$ consists of all the minimum elements from the $\mathcal{T}$-equivalence classes which partition $\mathcal{G} *_c t$. In contrast, for our purposes it is sufficient to construct a reduced set, which we denote by $\Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t)$, containing at least one of the minimum elements from each such $\mathcal{T}$-equivalence class; and that is what `generateConfigurations`$_{\mathcal{T}}$ does. Thus, if $\Theta_{\mathcal{T}}(\mathcal{G} *_c t)$ contains a_+ and a_- then $\Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t)$ must contain a_+ or a_- or possibly both.

Theorem 6.9.D: Assume the members of $\mathcal{T}$ are all adjacent transpositions. Then the refined algorithm, `generateConfigurations`$_{\mathcal{T}}$, yields $\Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t)$, a sufficiently large and sufficiently complete set of transpositionally canonical configurations. This set contains at least one minimum signed configuration from each $\mathcal{T}$-equivalence class whose members lie in $\mathcal{G} *_c t$. Formally, `generateConfigurations`$_{\mathcal{T}}$ $[t, \mathcal{S} \backslash \mathcal{T}]$ yields $\Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t)$, where $\Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t) \subseteq \Theta_{\mathcal{T}}(\mathcal{G} *_c t)$ and the sets of unsigned configurations are equal, that is, $\Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t)_{unsgined} = \Theta_{\mathcal{T}}(\mathcal{G} *_c t)_{unsgined}$.

Proof: A large portion of this proof parallels the proof of Theorem 6.5.A. Also it holds for the theoretical function $\theta_{\mathcal{T}}$, not just the special instantiation of it, $\Theta_{\mathcal{T}}$, hence we will present the proof in terms of $\theta_{\mathcal{T}}$.

Let $\mathcal{R} = \mathcal{S}\backslash\mathcal{T}$ be the set of non-transpositional generators. The refined algorithm, `generateConfigurations`$_{\mathcal{T}}$, in its n^{th} stage generates

$$\begin{aligned}\mathcal{F}_n &= \mathcal{F}_{n-1} \bigcup \mathcal{N}_{n-1}\\ \mathcal{N}_n &= \theta_{\mathcal{T}}[\mathcal{R} * \mathcal{N}_{n-1}]\backslash\mathcal{F}_n\end{aligned}$$

whereas in the original algorithm, `generateConfigurations`, we had $\mathcal{N}_n = \mathcal{S} * \mathcal{N}_{n-1}\backslash\mathcal{F}_n$. As in the proof of Theorem 6.5.A, since $\mathcal{G} *_c t$ is finite, the sequence $\mathcal{F}_1, \mathcal{F}_2, \ldots$ converges in a finite number of steps to its limiting value, say $\mathcal{F}$, the result of `generateConfigurations`$_{\mathcal{T}}[t, \mathcal{R}]$. We need to justify that $\mathcal{F}$ has the properties of $\Theta^{\ddagger}_{\mathcal{T}}(\mathcal{G} *_c t)$; that is, we need to prove that $\mathcal{F}$ contains at least one of the minimum elements from each of the $\mathcal{T}$-equivalence classes whose members make up $\mathcal{G} *_c t$. Formally, it remains to show that $\mathcal{F}_{unsgined} = \Theta_{\mathcal{T}}(\mathcal{G} *_c t)_{unsgined}$ and $\mathcal{F} \subseteq \Theta_{\mathcal{T}}(\mathcal{G} *_c t)$.

To adapt the original argument, let us define a new action, $*_\theta$, where $\rho *_\theta a = \theta_{\mathcal{T}}(\rho *_c a)$. In the first part of this proof, we will ignore signs completely, and only reinsert signs at the very last stage. Ignoring signs, $*_\theta$ is a group action on the minimum configurations $\theta_{\mathcal{T}}(C)$ since:

(*i*) $e *_\theta a = \theta_{\mathcal{T}}(e *_c a) = \theta_{\mathcal{T}}(a) = a$ for all $a \in \theta_{\mathcal{T}}(C)$.

(*ii*) $\sigma *_\theta \rho *_\theta a = \theta_{\mathcal{T}}(\sigma *_c \theta_{\mathcal{T}}(\rho *_c a)) = \pm\theta_{\mathcal{T}}(\sigma *_c \rho *_c a)$, by Lemma 6.9.A. But, $\theta_{\mathcal{T}}(\sigma *_c \rho *_c a) = \theta_{\mathcal{T}}(\sigma \cdot \rho *_c a) = (\sigma \cdot \rho) *_\theta a$. Thus, ignoring signs $\sigma *_\theta \rho *_\theta a = (\sigma \cdot \rho) *_\theta a$ for all $a \in \theta_{\mathcal{T}}(C)$ and $\sigma, \rho \in \mathcal{G}$

Thus, up to sign, $*_\theta$ is a group action of $\mathcal{G}$ on the minimum configurations $\theta_{\mathcal{T}}(C)$.

Now re-examine the proof of Theorem 6.5.A, which guaranteed that $\mathcal{F}_n$ converged to $\bigcup_i (\mathcal{S}^i *_c t)$, that is, $\mathcal{G} *_c t$. The reader should note that the original argument only depended on $*_c$ being a group action. Thus, in our new algorithm, by the exact same argument but replacing $\mathcal{S}$ by $\mathcal{R}$, $*_c$ by $*_\theta$, and t by $\theta_{\mathcal{T}}(t)$, it follows that $\mathcal{F}_n$ must converge to $\mathcal{F} = \bigcup_i (\mathcal{R}^i *_\theta \theta_{\mathcal{T}}(t))$ (remember, signs are being ignored). Moreover, it is easy to show that $\bigcup_i (\mathcal{R}^i *_\theta \theta_{\mathcal{T}}(t)) = \bigcup_i (\mathcal{S}^i *_\theta \theta_{\mathcal{T}}(t))$ by commuting transpositions through products of permutations, as was done in Lemma 6.9.A. But since $\mathcal{S}$ generates $\mathcal{G}$, using Lemma 6.9.A, we have $\bigcup_i (\mathcal{S}^i *_\theta \theta_{\mathcal{T}}(t)) = \bigcup_i \theta_{\mathcal{T}}(\mathcal{S}^i *_c t) = \theta_{\mathcal{T}}(\bigcup_i \mathcal{S}^i *_c t) = \theta_{\mathcal{T}}(\mathcal{G} *_c t)$. Therefore, we have shown $\mathcal{F}_{unsgined} = \theta_{\mathcal{T}}(\mathcal{G} *_c t)_{unsgined}$.

Now, reintroducing signs, we must show that $\mathcal{F}$ is a subset of $\theta_{\mathcal{T}}(\mathcal{G} *_c t)$. Consider any $a \in \mathcal{F}$. Then a is in some $\mathcal{N}_n = \theta_{\mathcal{T}}[\mathcal{R} * \mathcal{N}_{n-1}]\backslash\mathcal{F}_n$, hence $a = \theta_{\mathcal{T}}(s_1 *_c \theta_{\mathcal{T}}(s_2 *_c \ldots *_c \theta_{\mathcal{T}}(t) \ldots))$ for some $s_i \in \mathcal{R}$; but $s_1 *_c \theta_{\mathcal{T}}(s_2 *_c \ldots *_c \theta_{\mathcal{T}}(t) \ldots)$ must be in $\mathcal{G} *_c t$, hence $a \in \theta_{\mathcal{T}}(\mathcal{G} *_c t)$. Therefore, $\mathcal{F} \subseteq \theta_{\mathcal{T}}(\mathcal{G} *_c t)$. ■

This subsection concludes the presentation of the transpositional canonicalization subalgorithm. It is fully incorporated into the optimized tensor algorithm in §6.11 *The Optimized Algorithm*. Before we proceed onto the optimized algorithm, we must tackle the one important question remaining, that of zero-equivalence of a tensor. This is the topic of the next section.

6.10 Identically Zero Tensors

6.10.1 Zero Equivalence in the Basic Algorithm

For the theorems and proofs in this section, it is desirable to formalize the notation of an identically zero tensor.

Definition 6.10.A: A tensor or tensor product, say T, is *identically zero* if and only if $T = -T$ due to the symmetries of the tensors comprising T. Equivalently, T is identically zero if and only if $t \sim_{\mathcal{G}} -t$ by Definition 6.4.C (where t is the configuration of indices of the tensor product T and $\mathcal{G}$ is the signed permutation group of the symmetries of T).

Some readers may find the terminology *identically zero* somewhat strange, however it is strongly motivated from actual usage in physics. The tensor $\mathbf{R}^{a}{}_{abc}$ is zero by the symmetries of R alone. That is, $\mathbf{R}^{a}{}_{abc}$ vanishes in each and every physical situation independently of the metric or any other specific simplifications. Thus we say the tensor is *identically zero*. This is in contrast to tensors which may be zero under specific metrics or other conditions. For instance, the tensor $\mathbf{R}_{abcd}$ is zero in all flat metrics but is in general non-zero in curved metrics.

One of the consequences of our overall algorithm is that the test of whether the tensor being canonicalized is zero is extremely easy. The following theorems show that if a tensor is identically zero, then all signed configurations appearing in $\mathcal{G} *_c t$ must again appear in $\mathcal{G} *_c t$ with the opposite sign. In the case of our basic canonicalizing algorithm, all of $\mathcal{G} *_c t$ is generated, hence it is a simple matter to test whether or not any configuration appears with both signs, hence whether or not the tensor is identically zero.

Technical Note: For a signed configuration t, we will denote the oppositely signed configuration by $-t$, as one would expect. That is, if $t = c_{\pm}$ then $-t = c_{\mp}$, where c is the unsigned configuration part of t; and similarly for $-\sigma$ if σ is a signed permutation.

Theorem 6.10.A: A tensor or tensor product T is identically zero if and only if $\exists\, a \in C$ such that $a \in \mathcal{G} *_c t$ and $-a \in \mathcal{G} *_c t$, where t is the configuration of indices of the tensor product T.

Proof: ($\Longrightarrow$) If T is identically zero due to the symmetries of the tensors comprising T, then $t \sim_{\mathcal{G}} -t$, by Definition 6.10.A. Thus by definition, $-t = g *_c t$ for some $g \in \mathcal{G}$, hence, $-t \in \mathcal{G} *_c t$. Obviously $t \in \mathcal{G} *_c t$. So indeed $\exists\, a \in C$ such that $a \in \mathcal{G} *_c t$ and $-a \in \mathcal{G} *_c t$.

($\Longleftarrow$) Conversely, assume $a \in \mathcal{G} *_c t$ and $-a \in \mathcal{G} *_c t$ for some $a \in C$. Then $a \sim_{\mathcal{G}} t$, hence $t \sim_{\mathcal{G}} a$. Also $-a \sim_{\mathcal{G}} t$, hence $a \sim_{\mathcal{G}} -t$. Thus $t \sim_{\mathcal{G}} -t$, by transitivity. So T can be transformed to $-T$ by the symmetries of T, ergo T is identically zero. ■

We can immediately apply Theorem 6.10.A to create a function to determine whether the tensor product under consideration is identically zero. If removing the signs of the generated signed configurations and taking the union of this collection of unsigned configurations results in a smaller set, then obviously some of the signed configurations must only differ in their sign, and hence the overall tensor or tensor product must be identically zero. As in §6.7.1 *The Basic Canonicalization Algorithm*, we implement this idea with the following line of code.

```
In[1]:= tensorIdenticallyZeroQ @ signedConfigurations_List :=
          Union[sc[[2]] &sc /@ signedConfigurations]ten < signedConfigurationsten
```

Both for completeness and for use in the following subsection, we next show that if a tensor T having configuration t is identically zero, then every configuration reachable from t, that is, every configuration in $\mathcal{G} *_c t$, appears with both a plus and a minus in $\mathcal{G} *_c t$.

Theorem 6.10.B: If $\exists\, a \in \mathcal{G} *_c t$ such that $-a \in \mathcal{G} *_c t$, then $\forall\, b \in \mathcal{G} *_c t$ it is the case that $-b \in \mathcal{G} *_c t$.

Proof: Assume $a \in \mathcal{G} *_c t$ and $-a \in \mathcal{G} *_c t$. Then $t \sim_{\mathcal{G}} -t$ as shown in Theorem 6.10.A. Now consider any $b \in \mathcal{G} *_c t$. Then $b \sim_{\mathcal{G}} t \sim_{\mathcal{G}} -t$, hence $b \sim_{\mathcal{G}} -t$, hence $-b \sim_{\mathcal{G}} t$, hence $-b \in \mathcal{G} *_c t$. ■

Corollary 6.10.A: A tensor or tensor product T is identically zero if and only if $\mathcal{G} *_c t$ "double covers" up to sign the configurations reachable from t.

Incidentally, for some groups of symmetries associated with a given tensor, there is no configuration of indices for which the tensor will be identically zero, that is equal to zero by the symmetries alone. For example, say that $\mathcal{G}$ is generated only by positively signed permutations. Then it is clearly impossible to find any signed configuration t such that $-t \sim_{\mathcal{G}} t$.

6.10.2 Background to Zero Equivalence in the Optimized Algorithm

As indicated in the previous subsection, the simple code for `tensorIdenticallyZeroQ` suffices when we generate all configurations reachable from a configuration t, that is, $\mathcal{G} *_c t$. However, once we use transpositional canonicalization inside our optimized canonicalization algorithm, we will only generate a subset of $\mathcal{G} *_c t$. Specifically, $\texttt{generateConfigurations}_{\mathcal{T}}[t, \mathcal{S}]$ returns $\Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t) \subseteq \mathcal{G} *_c t$. This necessitates some simple modifications to the code of our "identically zero" testing function. Yet, quite some theoretical work is required to prove that these modifications are sound.

If a tensor is identically zero then, by the results in the previous subsection, we know that for each $a \in \mathcal{G} *_c t$ we must also have $-a \in \mathcal{G} *_c t$. In particular, for each $a \in \Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t)$ we must have $-a \in \mathcal{G} *_c t$, but it is not guaranteed that $-a \in \Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t)$. However, we do have the following result.

Theorem 6.10.C: Let T be a tensor or tensor product whose signed configuration is t. Then:

(*i*) If $-a \sim_{\mathcal{T}} a$ and/or $-a \in \Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t)$ for some $a \in \Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t)$, then T is identically zero.

(*ii*) If T is identically zero, then $-a \sim_{\mathcal{T}} a$ and/or $-a \in \Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t)$ for every $a \in \Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t)$.

Proof: Part (*i*) is trivial. For (*ii*) consider any $a \in \Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t)$. If T is zero, then we have $-a \in \mathcal{G} *_c t$ by Theorem 6.10.B, hence $-a$ is in some $\mathcal{T}$-equivalence class contained in $\mathcal{G} *_c t$. If a and $-a$ are in the same $\mathcal{T}$-equivalence then $-a \sim_{\mathcal{T}} a$. The other possibility is that a and $-a$ are in different $\mathcal{T}$-equivalence classes, say A and B. Then we easily show that $A = -B = \{-c \mid c \in B\}$. In detail, $c \in A \Leftrightarrow c \sim_{\mathcal{T}} a \Leftrightarrow -c \sim_{\mathcal{T}} -a \Leftrightarrow -c \in B \Leftrightarrow c \in -B$. Thus the configurations in A and B only differ in sign, and we know that $\prec_c$ does not distinguish between two configurations differing only in sign. Thus A and B will each have a unique minimum configuration, and these will differ only in their sign. But $a \in \Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t)$, hence a is the minimum element of A, so $-a$ must be the minimum element of B, hence $-a \in \Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t)$. ■

As a direct consequence of the previous theorem we can still determine if a tensor or tensor product is identically zero. This is accomplished by first testing to see if $\Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t)$ contains two signed configurations differing only in sign, and if not, then testing to see if $-a \sim_{\mathcal{T}} a$, where a is the canonical configuration $min_c(\Theta^{\dagger}_{\mathcal{T}}(\mathcal{G} *_c t))$. That is, use `tensorIdenticallyZeroQ` as before to test for a double presence and introduce a piece of code to test for $-a \sim_{\mathcal{T}} a$. Our goal now is to find an efficient way to determine whether $-a \sim_{\mathcal{T}} a$, for any given signed configuration a. The following subsections work towards this by first introducing "induced" transpositions, then proving several theorems involving them, before finally presenting an algorithm for the efficient determination of whether a tensor is identically zero.

6.10.3 Induced Transpositions

Let us now introduce the notion of an induced transposition, which play a pivotal role in determining whether $-a \sim_{\mathcal{T}} a$.

Definition 6.10.B: Every transposition τ acting on just summed indices of a configuration $a \in C$ is equivalent to an *induced transposition* σ acting on the same configuration a, that is, $\tau *_c a = \sigma *_c a$. Formally, if $\tau = (m \leftrightarrow n)$ and $a_m = s_{m,\overline{m}}$ and $a_n = s_{n,\overline{n}}$, then $\sigma = (\overline{m} \leftrightarrow \overline{n})$ is the induced transposition of τ relative to the configuration a. We will denote this by $induced_a(\tau)$, or by $\overline{\tau}$ when no confusion is possible.

Consider the symmetry $(1 \leftrightarrow 2)_+$ of the tensor T acting on the configuration generated by $\mathbf{T}^{abc}{}_{bac}$. It is equivalent to the induced transposition $(4 \leftrightarrow 5)_+$ acting on the same configuration.

```
In[2]:= (1 ↔ 2)+ *c ⟨T^abc_bac⟩c == (4 ↔ 5)+ *c ⟨T^abc_bac⟩c

Out[2]= True
```

It is easy to see that a "real" symmetry, like the symmetric transposition on the first and second indices of the tensor T, is fundamentally different from an induced transposition. For instance, as above, $(1 \leftrightarrow 2)_+$ together with the tensor configuration $\mathbf{T}^{abc}{}_{bac}$ induces the transposition $(4 \leftrightarrow 5)_+$ whereas $(1 \leftrightarrow 2)_+$ together with the tensor configuration $\mathbf{T}^{ab\ c}{}_{a\ cb}$ induces the transposition $(3 \leftrightarrow 6)_+$. This clearly shows that the validity of using an induced transposition is dependent on the specific configuration of the indices, unlike the transposition $(1 \leftrightarrow 2)_+$ which can be validly used on any signed configuration since it is one of the symmetries of T. In summary, an induced transposition is only valid for a specific configuration.

By examining the configuration we can see that the induced transposition is just the image of the original transposition under the map taking each summed position of the configuration to its corresponding summed index position. To illustrate this point let us examine the configuration in the example above.

```
In[3]:= ⟨T^abc_bac⟩c

Out[3]= {s1,5, s2,4, s3,6, s4,2, s5,1, s6,3}+
```

We can see that positions 1 and 5 are linked, positions 2 and 4 are linked, and positions 3 and 6 are linked. We can put this linking into an algorithmic form by generating a set of rules taking each summed position to its summed partner's position.

```
In[4]:= inducingRules = Flatten @ Cases[%[2], si_,j_ → {i → j}]

Out[4]= {1 → 5, 2 → 4, 3 → 6, 4 → 2, 5 → 1, 6 → 3}
```

Observe that *inducingRules* is derived solely from the configuration and has nothing to do with which transpositional symmetries we might apply to the configuration. We can now easily create a simple function that returns the induced transposition of a given transposition under a given configuration.

```
In[5]:= Symbolize[iind]; Symbolize[jind]

In[6]:= inducedTransposition[(i_ ↔ j_)sign_] :=
          Block[
            {iind = i /. inducingRules,
             jind = j /. inducingRules},
            If[iind < jind, (iind ↔ jind)sign, (jind ↔ iind)sign]]
```

Let us use this function to verify that indeed the above $(1 \leftrightarrow 2)_+$ induces the transposition $(4 \leftrightarrow 5)_+$ for the given configuration.

```
In[7]:= inducedTransposition[(1 ↔ 2)+]

Out[7]= (4 ↔ 5)+
```

We will have occasion to use sets of induced transpositions. That is, given a configuration $a \in C$ and a set of transpositions $\mathcal{T}$, then $induced_a(\mathcal{T})$ will be the set of all transpositions induced from $\mathcal{T}$ under the configuration a. Formally, $induced_a(\mathcal{T}) = \{induced_a(\tau) \mid \tau \in \mathcal{T}\}$.

6.10.4 Theorems about Zero Equivalence in the Optimized Algorithm

We are now in a position to state and prove the major results underlying the efficient determination of the following question. Is $-a \sim_{\mathcal{T}} a$?

Definition 6.10.C: The *closure of a set of signed transpositions* $\mathcal{T}$ is the set of all signed transpositions in the signed group generated by $\mathcal{T}$, that is, all signed transpositions in $\langle \mathcal{T} \rangle$. A set of transpositions $\mathcal{T}$ is *closed* if $closure(\mathcal{T}) = \mathcal{T}$. We will usually denote the closure of a set of transpositions $\mathcal{T}$ by $\mathcal{T}_\blacksquare$

Technical Note: Under other circumstances, we might have used $\overline{\mathcal{T}}$ for the closure, but this can also be read $\mathcal{T}$ conjugate. Moreover, $\overline{\mathcal{T}}$ looks too much like the induced transpositional set. Consequently, we have chosen the notation $\mathcal{T}_\blacksquare$ for the closure of a set of transpositions.

Working with a closed set of signed transpositions will be quite important in this subsection and in the optimized canonicalizing algorithm. In fact, when the symmetries for a primitive tensor are initially specified, internally the package will immediately generate the closure of the given set of signed transpositions, since this closure is made use of every time we canonicalize a tensor product involving this primitive tensor. However, it should be noted that even though the closure of a set $\mathcal{T}$ of transpositions contains all possible transpositions reachable from $\mathcal{T}$, the closure $\mathcal{T}_\blacksquare$ does not form a group.

Lemma 6.10.A: Given a set of signed transpositions $\mathcal{T}$, if $\exists\, \sigma_i \in \mathcal{T}$ such that $\prod_i \sigma_i = -\mathbb{I}$ then there exists an unsigned transposition τ such that both of the signed transpositions τ_+ and τ_- are in $\mathcal{T}_\blacksquare$.

Proof: Actually we prove a slightly stronger result, namely that for any unsigned transposition τ, we have $\tau_- \in \mathcal{T}_\blacksquare$ if and only if $\tau_+ \in \mathcal{T}_\blacksquare$. This follows from the fact that if $\tau' \in \mathcal{T}_\blacksquare$, then $-\tau' = -\mathbb{I} \cdot \tau' = (\prod_i \sigma_i) \cdot \tau'$, hence $-\tau'$ is a transposition in $\langle \mathcal{T} \rangle$, hence $-\tau' \in \mathcal{T}_\blacksquare$ since $\mathcal{T}_\blacksquare$ is closed. ■

Unfortunately just examining all transpositions in $\mathcal{T}_\blacksquare$ will not always tell us when a configuration is identically zero. Let us quickly give a motivating example of this behavior. Consider the

tensor $\mathbf{T}^{i\,j}_{\;i\,j}$ and the corresponding signed configuration $t = \overline{\{s_{1,3},\, s_{2,4},\, s_{3,1},\, s_{4,2}\}}_+$. Furthermore, let us assume this tensor T has the symmetries $\tau_1 = (1 \leftrightarrow 2)_-$ and $\tau_2 = (3 \leftrightarrow 4)_+$. Clearly no product or combination of products of τ_1 and τ_2 are going to equal $-\mathbb{I}$. However $\tau_1 *_c (\tau_2 *_c t)$ is equivalent to $-t$.

```
In[8]:= (1 ↔ 2)_ *c ((3 ↔ 4)+ *c {s1,3, s2,4, s3,1, s4,2}+)
Out[8]= {s1,3, s2,4, s3,1, s4,2}-
```

This demonstrates that there may exist some $\tau_i \in \mathcal{T}$ such that $(\prod_i \tau_i) *_c a = -a$, whereas there is no set $\tau_i \in \mathcal{T}$ such that $(\prod_i \tau_i) = -\mathbb{I}$. The underlying reason for this is that the group action entangles the permutation operation with the configuration relabeling, and in some circumstances they can interact to yield an overall negative sign.

Before trying to find conditions for when $(\prod_i \tau_i) *_c a = -a$ for some $\tau_i \in \mathcal{T}$, let us first consider the special case when $\tau *_c a = -a$. For an example of this case, we have the following:

```
In[9]:= signedConfig = ⟨X^{i a}_{i b}⟩c
Out[9]= {s1,3, a2↑, s3,1, b4↓}+
In[10]:= (1 ↔ 3)_ *c signedConfig
Out[10]= {s1,3, a2↑, s3,1, b4↓}-
```

Let us embody this in the following lemma.

Lemma 6.10.B: For any signed transposition τ and signed configuration $a \in C$, we have $\tau *_c a = -a$ if and only if $\tau = (m \leftrightarrow \overline{m})_-$ for some m such that a_m, the index at position m in the signed configuration a, is $s_{m,\overline{m}}$.

Proof: ($\Longleftarrow$) is trivial. For ($\Longrightarrow$) assume $\tau *_c a = -a$ and say $\tau = (m \leftrightarrow n)_-$. (Clearly $\tau = (m \leftrightarrow n)_+$ would not do.) We first show that the indices in a at positions m and n must be summed indices. For if one of them, say the index at m, were free, then it would be moved to position n in $\tau *_c a$, but it appears only in position m in $-a$, contradicting $\tau *_c a = -a$.

So say $a_m = s_{m,\overline{m}}$ and $a_n = s_{n,\overline{n}}$. If m, $\overline{m}$, n and $\overline{n}$ are all distinct, then the index at position m in $a' = \tau *_c a$ is $a_{m,\overline{n}}$; but it is also $a_{m,\overline{m}}$ since $a' = -a$, hence $\overline{m} = \overline{n}$, hence $m = n$, which is not possible if $(m \leftrightarrow n)$ is to be a transposition.

So m, $\overline{m}$, n and $\overline{n}$ are not all distinct. But $n \neq m$ and $n \neq \overline{n}$, hence this leaves only the possibility that $n = \overline{m}$, hence also $m = \overline{n}$. Then $a_m = s_{m,\overline{m}}$ as before and $a_n = s_{n,\overline{n}} = s_{\overline{m},m} = a_{\overline{m}}$; and indeed one can easily check that $(m \leftrightarrow \overline{m})_- *_c a = -a$. ■

It transpires that our main theorem, which states conditions for when $-a \sim_{\mathcal{T}} a$, will have to involve not only all members of $\mathcal{T}$ but also all of the induced transpositions relative to a, that is, all members of $induced_a(\mathcal{T})$. The proof of this main theorem depends critically on the following main lemma.

Lemma 6.10.C: Say $\mathcal{T}_\blacksquare$ is a closed set of signed transpositions. Consider any product $\prod_{i=1}^{k} \tau_i = \tau_k \cdot \ldots \cdot \tau_2 \cdot \tau_1$, with $\tau_i \in \mathcal{T}_\blacksquare$ and consider any integer m appearing in one of the τ_i. Then $\exists\, \tau'_i \in \mathcal{T}_\blacksquare$ such that $\prod_{i=1}^{k'} \tau'_i = \prod_{i=1}^{k} \tau_i$, where $k' \leqslant k$ and m can occur only in the rightmost τ'_i, if at all; and furthermore, only the positions moved by the individual τ_i are moved by the τ'_i.

Proof: Consider the product of signed transpositions $\prod_{i=1}^{k} \tau_i = \tau_k \cdot \ldots \cdot \tau_2 \cdot \tau_1$ and consider an m appearing in one of the τ_i. Find the *leftmost* occurrence of m in the product. If this occurrence is in τ_1, then we are done. So assume it is in $\tau_{j+1} = (m \leftrightarrow x)_\pm$ and consider the product $\tau_{j+1} \cdot \tau_j$ within $\prod_{i=1}^{k} \tau_i$. If we ignore signs, then there are only four possible cases for $\tau_{j+1} \cdot \tau_j$.

$\tau_{j+1} \cdot \tau_j$	=	$\tau'_{j+1} \cdot \tau'_j$
$(m \leftrightarrow x) \cdot (m \leftrightarrow x)$	=	$\mathbb{I}$
$(m \leftrightarrow x) \cdot (m \leftrightarrow y)$	=	$(x \leftrightarrow y) \cdot (m \leftrightarrow x)$
$(m \leftrightarrow x) \cdot (y \leftrightarrow z)$	=	$(y \leftrightarrow z) \cdot (m \leftrightarrow x)$
$(m \leftrightarrow x) \cdot (x \leftrightarrow y)$	=	$(x \leftrightarrow y) \cdot (m \leftrightarrow y)$

(6.10.a)

In the first case we will have reduced the number of transpositions by two. In the three remaining cases, the leftmost occurrence of m has moved one place further right. Also, we have not increased the number of transpositions involved, and any of the new transpositions only move positions moved by the old transpositions. Moreover, any possible new transpositions are in $\mathcal{T}_\blacksquare$, since $\mathcal{T}_\blacksquare$ is closed. For example, in the second case, multiplying both sides on the right by $(m \leftrightarrow x)$ we can see that $(x \leftrightarrow y)$ is just $(m \leftrightarrow x) \cdot (m \leftrightarrow y) \cdot (m \leftrightarrow x)$, hence is in $\mathcal{T}_\blacksquare$.

Repeating the above steps as many times as possible will result in either a single transposition on the far right involving m or no transpositions involving m, as stated. ■

Theorem 6.10.D: Given a set of signed transpositions $\mathcal{T}$ and a signed configuration $a \in C$, then $-a \sim_{\mathcal{T}} a$ if and only if there exists a signed transposition $\tau \in \mathcal{T}_\blacksquare$ such that either $\tau *_c a = -a$ or both τ and $-\tau$ are present in $\mathcal{T}_\blacksquare \bigcup \mathit{induced}_a(\mathcal{T}_\blacksquare)$.

Proof: ($\Longrightarrow$) If $-a \sim_{\mathcal{T}} a$ then there exists $\tau_i \in \mathcal{T}$ such that $(\prod_i \tau_i) *_c a = -a$. If $\prod_i \tau_i = -\mathbb{I}$ then by Lemma 6.10.A our present theorem is obviously satisfied. Therefore, assume that $\prod_i \tau_i \neq -\mathbb{I}$ yet $\prod_i \tau_i *_c a = -a$. Furthermore, assume there is more than one transposition in the product $\prod_i \tau_i$; for if not then we would have a $\tau = \tau_1 \in \mathcal{T}$ such that $\tau *_c a = -a$, which would satisfy the theorem.

Now pick any position that is moved in the transpositions, say position m. Find the leftmost transposition that moves m and then successively "move" this transposition further to the right. We perform this exactly as Lemma 6.10.C allows us. Either we encounter a transposition that is present in both signs, thus satisfying the theorem; or we eliminate the transpositions involving m, in which case we pick a new m and repeat the whole procedure; or we obtain a new product of transpositions, with a single rightmost transposition involving m and some other position, say n.

At this stage we have $\prod_{i=1}^{k} \tau_i = (\prod_{i=1}^{k'} \tau'_i) \cdot (m \leftrightarrow n)_{\pm}$ where the $\tau'_i \in \mathcal{T}_{\blacksquare}$ do not involve the position m. The indices at both positions m and n must be summed indices or else we arrive at a contradiction. For instance, say that the index at position m in a is a free index f. Then in $a' = (m \leftrightarrow n)_{\pm} *_c a$ the free index f has moved to position n in a'. Since the product $\prod_{i=1}^{k'} \tau'_i$ does not move m (by construction), then applying the product $\prod_{i=1}^{k'} \tau'_i$ to a' cannot move the index f back to position m. Because the position of f in $(\prod_{i=1}^{k'} \tau'_i) \cdot (m \leftrightarrow n)_{\pm} *_c a$ cannot be m, clearly $(\prod_{i=1}^{k'} \tau'_i) \cdot (m \leftrightarrow n)_{\pm} *_c a \neq -a$.

Since the indices at both positions m and n must be summed, they are of the form $s_{m,\overline{m}}$ and $s_{n,\overline{n}}$. If it transpires that the m, $\overline{m}$, n, $\overline{n}$ are not distinct, then since $m \neq \overline{m}$ and $n \neq \overline{n}$ it follows that $(m \leftrightarrow n)_{\pm}$ must in fact be equal to $(m \leftrightarrow \overline{m})_{\pm}$. If the latter is $(m \leftrightarrow \overline{m})_{+}$ then $(m \leftrightarrow \overline{m})_{+} *_c a = a$, and we can eliminate the transposition. If it is $(m \leftrightarrow \overline{m})_{-}$ then $(m \leftrightarrow \overline{m})_{-} *_c a = -a$ by Lemma 6.10.B, and we have satisfied the theorem.

Progressing on, we now have $\prod_{i=1}^{k} \tau_i = (\prod_{i=1}^{k'} \tau'_i) \cdot (m \leftrightarrow n)_{\pm}$, where the $\tau'_i \in \mathcal{T}_{\blacksquare}$ do not involve the position m, where $k' < k$, and where the summed indices at positions m and n in a are $s_{m,\overline{m}}$ and $s_{n,\overline{n}}$ with m, $\overline{m}$, n, $\overline{n}$ all distinct. Let us temporarily replace $s_{m,\overline{m}}$ and $s_{n,\overline{n}}$ by the fixed dummy labels α and β. To do so gives results equivalent to working with the $s_{i,j}$ notation and also makes the upcoming explanation easier to understand. (By our whole design, we can equivalently interchange our fixed dummy labels in our configurations with $s_{i,j}$ labels and vice versa. However, if one wishes, one can maintain the $s_{i,j}$ labels and painfully verify the same overall result.)

Now consider $(m \leftrightarrow n)$ acting on a by examining the movement of the indices at the distinct positions m, $\overline{m}$, n and $\overline{n}$. Ignoring the sign, $(m \leftrightarrow n) *_c a$ gives the following.

	m	n	$\overline{m}$	$\overline{n}$
$m \leftrightarrow n$	α	β	α	β
	β	α	α	β

(6.10.b)

Since the τ'_i in $\prod_{i=1}^{k'} \tau'_i$ do not involve m, this implies that the m^{th} index of $(\prod_{i=1}^{k'} \tau'_i) \cdot (m \leftrightarrow n)_{\pm} *_c a$ must remain fixed as β. However, the overall product of transpositions acting on a results in $-a$, hence after all the transpositions have been applied the m^{th} index must still be summed with the $\overline{m}^{th}$ index. Therefore, we know that the $(\prod_{i=1}^{k'} \tau'_i)$ must move the complementary dummy index β to the position $\overline{m}$. Since $\overline{m}$ is moved by the product of transpositions, let us repeat the "successive shifting to the right" procedure, allowed by Lemma 6.10.C, for the leftmost transposition in $\prod_{i=1}^{k'} \tau'_i$ that moves $\overline{m}$. In performing this shifting we either encounter a transposition that is present in both signs, thus satisfying the theorem; or we obtain a new product of transpositions, with a single rightmost transposition involving $\overline{m}$ and some other position, say p.

We now have $\prod_{i=1}^{k} \tau_i = (\prod_{i=1}^{k''} \tau''_i) \cdot (\overline{m} \leftrightarrow p)_{\pm} \cdot (m \leftrightarrow n)_{\pm}$, where the $\tau''_i \in \mathcal{T}_{\blacksquare}$ do not involve either m or $\overline{m}$ and where $k'' \leqslant k-2$. It is clear that only $(m \leftrightarrow n)$ and $(\overline{m} \leftrightarrow p)$ involve the positions m and $\overline{m}$ (since none of the τ''_i move m or $\overline{m}$). However, after $(\overline{m} \leftrightarrow p)_{\pm} \cdot (m \leftrightarrow n)_{\pm}$ operates on a, the m^{th} index must still be summed with the $\overline{m}^{th}$ index. Thus, it must be the case that $(\overline{m} \leftrightarrow p)$ moves the complementary dummy

index β (which was at $\overline{n}$) to $\overline{m}$, hence p must be $\overline{n}$. Hence the product must narrow to $\prod_{i=1}^{k} \tau_i = (\prod_{i=1}^{k''} \tau_i'') \cdot (\overline{m} \leftrightarrow \overline{n})_{\pm} \cdot (m \leftrightarrow n)_{\pm}$. Continuing with our temporary replacement of $s_{m,\overline{m}}$ and $s_{n,\overline{n}}$ by the fixed dummy labels α and β, let us examine $(\overline{m} \leftrightarrow \overline{n}) \cdot (m \leftrightarrow n)$ acting on a by tracking the movement of the indices at the distinct positions m, $\overline{m}$, n and $\overline{n}$.

	m	n	$\overline{m}$	$\overline{n}$
$m \leftrightarrow n$	α	β	α	β
$\overline{m} \leftrightarrow \overline{n}$	β	α	α	β
	β	α	β	α

(6.10.c)

Thus, since we relabel after applying each transposition, from (6.10.c) we can see that we must have $(\overline{m} \leftrightarrow \overline{n}) \cdot (m \leftrightarrow n) *_c a = a_{\pm}$. If the signs of $(\overline{m} \leftrightarrow \overline{n})$ and $(m \leftrightarrow n)$ are the same, then we have eliminated another pair of transpositions. If the signs are different then clearly, since the induced image of either one of these transpositions under the configuration a is the other, we have found a transposition such that both signs of the transposition are elements of $\mathcal{T}_{\blacksquare} \bigcup induced_a(\mathcal{T}_{\blacksquare})$, so we have satisfied the theorem. For example, if the factors were $(\overline{m} \leftrightarrow \overline{n})_-$ and $(m \leftrightarrow n)_+$ then the induced image of $(m \leftrightarrow n)_+$ under the configuration a is $(\overline{m} \leftrightarrow \overline{n})_+$, hence $(\overline{m} \leftrightarrow \overline{n})_+$ and $(\overline{m} \leftrightarrow \overline{n})_-$ are both members of our joined set.

At each stage, our process either reduces the number of transpositions or finds a pair of transpositions that satisfies the theorem, that is, finds a τ and $-\tau$ which are both in $\mathcal{T}_{\blacksquare} \bigcup induced_a(\mathcal{T}_{\blacksquare})$. Say that at no stage do we find a pair of transpositions that satisfies the theorem. Then we must terminate at a stage with less than two transpositions remaining, or else another stage of our process could be performed. Clearly, we cannot terminate with zero transpositions since $\mathbb{I} *_c a \neq -a$. Therefore, we must terminate with one transposition, that is, we terminate with $\tau *_c a = -a$. But then we have satisfied the theorem because the resulting τ will be in $\mathcal{T}_{\blacksquare}$ since the initial τ_i were, and since $\mathcal{T}_{\blacksquare}$ is closed.

($\Longleftarrow$) If $\tau *_c a = -a$ for some $\tau \in \mathcal{T}_{\blacksquare}$, then unquestionably we have $a \sim_{\mathcal{T}} -a$. If both τ and $-\tau$ are present in $\mathcal{T}_{\blacksquare} \bigcup induced_a(\mathcal{T}_{\blacksquare})$, then we have several cases to consider. Without loss of generality assume $\tau = \sigma_+$ and $-\tau = \sigma_-$. If σ_+ and σ_- are both in $\mathcal{T}_{\blacksquare}$ then the result is obvious. If say $\sigma_+ \in \mathcal{T}_{\blacksquare}$ and $\sigma_- \in induced_a(\mathcal{T}_{\blacksquare})$ then σ_+ and $\overline{\sigma}_-$ are both in $\mathcal{T}_{\blacksquare}$, where $\overline{\sigma}_- = induced_a(\sigma_-)$, since $induced_a(induced_a(\mathcal{T}_{\blacksquare})) = \mathcal{T}_{\blacksquare}$. Also, $(\sigma_+ \cdot \overline{\sigma}_-) *_c a = \sigma_+ *_c (\overline{\sigma}_- *_c a) = \sigma_+ *_c (\sigma_- *_c a) = -a$, hence $\exists\, \tau_i \in \mathcal{T}$ such that $\prod_i \tau_i * a = -a$, hence $a \sim_{\mathcal{T}} -a$. Finally, if both σ_+ and σ_- are in $induced_a(\mathcal{T}_{\blacksquare})$, then both $\overline{\sigma}_+$ and $\overline{\sigma}_-$ are in $\mathcal{T}_{\blacksquare}$ and $(\overline{\sigma}_+ \cdot \overline{\sigma}_-) *_c a = -\mathbb{I} *_c a = -a$, hence again $\exists\, \tau_i \in \mathcal{T}$ such that $\prod_i \tau_i *_c a = -a$. ■

To apply this theorem consider our preceding example of the tensor $t = \left\langle \mathbf{T}^{i\,j}{}_{i\,j} \right\rangle_c$ together with the symmetries $\tau_1 = (1 \leftrightarrow 2)_-$ and $\tau_2 = (3 \leftrightarrow 4)_+$. As was shown previously $\tau_1 *_c (\tau_2 *_c t) = -t$, and clearly, no product of τ_1 and τ_2 could possibly be equal to $-\mathbb{I}$. However if we examine $\{\tau_1, \tau_2\} \bigcup \{\overline{\tau_1}, \overline{\tau_2}\}$, we should be able to find oppositely signed transpositions in this set, and indeed we do.

```
In[11]:= 𝒯 = {(1 ↔ 2)_, (3 ↔ 4)_+}; signedConfig = ⟨T^{i j}_{i j}⟩_c;
         inducingRules = Flatten @ Cases[signedConfig_[[2]], s_i_, j_ → {i → j}];

In[13]:= 𝒯 ⋃ inducedTransposition /@ 𝒯
Out[13]= {(1 ↔ 2)_, (3 ↔ 4)_, (1 ↔ 2)_+, (3 ↔ 4)_+}
```

6.10.5 Zero Equivalence in the Optimized Algorithm

Finally, we can now apply the theorems developed in the forgoing subsections. First, we calculate all of the transpositions in the closure of our given set of transposition symmetry generators, that is, we calculate $\mathcal{T}_\blacksquare$ from $\mathcal{T}$. We then calculate the set of transpositions this induces. Lastly, we determine if there are any transpositions which move the same elements but have a different sign. If so we have shown $-a \sim_{\mathcal{T}} a$. If not we know that $-a \not\sim_{\mathcal{T}} a$. Here is our new overall algorithm for testing for zero equivalence. It fundamentally relies on the results of Corollary 6.10.A, Theorem 6.10.D, and Lemma 6.10.B.

```
In[14]:= Symbolize[𝒯■]

In[15]:= tensorIdenticallyZeroQ[minSignedConfiguration_,
             signedConfigurations_List, 𝒯■ : _List] :=
         Block[
             {allT, minConfiguration = minSignedConfiguration[[2]], inducingRules},
             inducingRules = Flatten @ Cases[minConfiguration, s_i_, j_ → {i → j}];
             allT = 𝒯■ ⋃ (inducedTransposition /@ 𝒯■);
             (Union[sc[[2]] &sc /@ signedConfigurations]_len < signedConfigurations_len) ⋁
                 (Union[sT[[2]] &sT /@ allT]_len < allT_len) ⋁
                 directlyZeroQ[minConfiguration, 𝒯■]]
```

The code directly above uses the ancillary function `directZeroQ` to determine if the tensor product under consideration is zero directly from a single transposition in the closure of $\mathcal{T}$. Formally, `directZeroQ` determines if $\exists\, \tau \in \mathcal{T}_\blacksquare$ such that $\tau *_c t = -t$ where t is the minimum signed configuration found previously.

```
In[16]:= directlyZeroQ[minConfiguration_, 𝒯■ : _List] :=
             Block[{reducedIndices = reduceIndex /@ minConfiguration},
                 Or @@ zeroTestAux /@ 𝒯■]

In[17]:= zeroTestAux @ (i_ ↔ j_)_+ = False;
         zeroTestAux @ (i_ ↔ j_)_ := reducedIndices[[i]] ≡ reducedIndices[[j]]

In[19]:= reduceIndex[s_i_, j_] := s @ Min[i, j];
         reduceIndex @ h_[index_, _] := h @ index
```

Technical Note: Our code must also cover the case when the tensor product under consideration has indices which contain coordinates, for instance, $\mathbf{R}_{0\,0\,\alpha\,\beta}$.

We can demonstrate this code in operation with the following examples. We use the function `OptimizedCanonicalize` which is more advanced than our `BasicCanonicalize`, yet still lacks several refinements that we will develop later. The function `OptimizedCanonicalize` is presented in §6.11.3 *The Optimized Canonicalization Algorithm.*

```
In[21]:= OptimizedCanonicalize[R^a_acd]
Out[21]= 0
In[22]:= OptimizedCanonicalize[R^00_cd]
Out[22]= 0
In[23]:= OptimizedCanonicalize[R^αθ_ρρ]
Out[23]= 0
In[24]:= OptimizedCanonicalize[R^αθ_ρρ;β]
Out[24]= 0
In[25]:= OptimizedCanonicalize[R^α_γρθ;β R^γ_α]
Out[25]= 0
In[26]:= OptimizedCanonicalize[R^αθ_ρρ,β]
Out[26]= 0
In[27]:= OptimizedCanonicalize[R^ab_ac R^c_bef]
Out[27]= 0
In[28]:= OptimizedCanonicalize[R^ab_cd S^e_abf]
Out[28]= 0
In[29]:= OptimizedCanonicalize[R^ab_ac R^f_def A^ec_b]
Out[29]= 0
In[30]:= OptimizedCanonicalize[R^abcd S^mn_ba]
Out[30]= 0
```

Given the length of the proof for Theorem 6.10.D, one naturally feels that there may be simpler methods to determine whether a tensor is identically zero. Indeed, intuition tells us that it should be possible, using a cardinality equation to tell if the tensor is zero-equivalent by comparing how many classes there should be with how many we actually manage to find. Such

an approach would at least be conceptually simpler. However, it would be computationally more expensive since our implementation of `tensorIdenticallyZeroQ` is actually very fast.

Further to the above comment, there is an obvious and conceptually simpler alternative to the method provided by Theorem 6.10.D for determining if $-a \sim_{\mathcal{T}} a$. This is to generate all possible signed configurations which are $\mathcal{T}$-equivalent to a. This is easily done using our original closure method, `generateConfigurations`, of §6.5.2 *The Algorithm for Generating Configurations*, but applied to $\mathcal{T}$ rather than the full symmetry generator set $\mathcal{S}$.

Considering that there exists such a conceptually simple procedure, why did we develop the elaborate machinery of §6.10.3 *Induced Transpositions* and §6.10.4 *Theorems about Zero Equivalence in the Optimized Algorithm*? The reason is that comparing induced transpositions will be faster. Without entering into detail, our more complex method above for transpositions having n indices scales according to n^2, whereas generating all configurations in the $\mathcal{T}$-equivalence class scales as $(n/2)!$ In addition, the same sorts of concepts involved in determining if $-a \sim_{\mathcal{T}} a$ will be extensively used later in §6.13 *Refinements for Mixed Index Classes*.

Let us now combine the developments of the last three sections to form our optimized canonicalized algorithm.

6.11 The Optimized Algorithm

In this section we apply the theories and ideas developed in the last three sections, §6.8 *Generators and Group Theoretic Underpinnings*, §6.9 *Transpositional Canonicalization*, and §6.10 *Identically Zero Tensors*, to culminate in our optimized canonicalization algorithm. This algorithm incorporates the canonicalization of the free indices and the subsequent stabilization of the permutation group on the free indices, transpositional canonicalization, and the identification of zero tensors.

In §6.12 *Refinements for Partial Derivatives*, we extend the algorithm to include summed indices with fixed elevations, which arise when dealing with partial derivatives. Then, succeeding this in §6.13 *Refinements for Mixed Index Classes*, we make a further extension to allow summed indices from mixed index classes. Finally in §6.14 *Linear Symmetries and the Complete Algorithm*, we incorporate linear symmetries and all of the previous theories, algorithms, and developments, into an overarching canonicalization algorithm.

6.11.1 Canonicalization of Free Indices

After the group theoretic developments in §6.8 *Generators and Group Theoretic Underpinnings* and the presentation of the transpositional canonicalization results in §6.9 *Transpositional Canonicalization*, we are ready to apply these to the canonicalization of the free indices of a configuration. Our goal is to transform an initial configuration by the permutations in its signed permutation group in such a way as to ensure that all of the free indices in the returned configuration are in their canonical positions. The method presented in this subsection entails the use of a "downhill" method requiring a set of JRDES generators — cf. §6.8.5 *Jointly Recursively Directional and Extrema Stabilizing (JRDES) Sets.*

Definition 6.11.A: A free index α is in its *canonical position* in a signed configuration a if its position in a is the same as its position in the final canonical configuration.

Given a signed configuration and a JRDES set of generators, say $\mathcal{S}$, for a permutation group $\mathcal{G}$, the following algorithm uses a "downhill" method to compute an equivalent signed configuration in which all the free indices are in their canonical positions. The algorithm requires that $\mathcal{S}$ be split into its transpositional, its complex, and its degenerate parts, that is $\mathcal{S}_{\mathcal{T}}$, $\mathcal{S}_{\mathcal{X}}$, and $\mathcal{S}_{\mathcal{D}}$ — cf. §6.8 *Generators and Group Theoretic Underpinnings*.

```
In[1]:= canonicalizeFreeIndices[
            sc_signedConfiguration, S_T : _, S_X : _, S_D : _] :=
        Block[{new = sc, final = Null, temp},
            While[final ≠ new,
                final = new;
                If[(temp = ρ *_c new) <_c new, new = temp] &_ρ /@ S_D;];
            final = Null;
            While[final ≠ new,
                final = new = Θ_{S_T}[new];
                If[(temp = ρ *_c new) <_c new, new = temp] &_ρ /@ S_X];
            final]
```

In a sense this algorithm canonicalizes the free indices in parallel since it applies any generator in any place that makes the configuration smaller. Let us illustrate the algorithm with a few simple examples and then later proceed onto the validation of the algorithm. Consider the following signed configuration and its symmetries.

```
In[2]:= signedConfig = ⟨R^{mpaq} R^{cn}_{pm} R_{qbn}^{d}⟩_c
```

$$\overline{\{s_{1,8},\ s_{2,7},\ a^{\uparrow}_{3},\ s_{4,9},\ c^{\uparrow}_{5},\ s_{6,11},\ s_{7,2},\ s_{8,1},\ s_{9,4},\ b^{\downarrow}_{10},\ s_{11,6},\ d^{\uparrow}_{12}\}}_{+}$$

```
In[3]:= {S_T, S_X, S_D} = ⟨R^{mpaq} R^{cn}_{pm} R_{qbn}^{d}⟩_{S_T,X,D};
```

```
In[4]:= canonicalizeFreeIndices[signedConfig, S_T, S_X, S_D]
```

$$\overline{\{a^{\uparrow}_{1},\ s_{2,11},\ s_{3,7},\ s_{4,8},\ c^{\uparrow}_{5},\ s_{6,10},\ s_{7,3},\ s_{8,4},\ d^{\uparrow}_{9},\ s_{10,6},\ s_{11,2},\ b^{\downarrow}_{12}\}}_{+}$$

Often after canonicalizing the free indices, we will be close to the optimal form of the tensor. This is especially true in the case when there are enough free indices to restrict the permutations which can subsequently be used. In fact, for the example above, it is a simple matter to confirm that just canonicalizing the free indices yields the minimum configuration.

```
In[5]:= equivalentConfigurations = generateConfigurations[signedConfig, S_T ⋃ S_X ⋃ S_D];

In[6]:= Min_c @ equivalentConfigurations

Out[6]= {a_1^↑, S_{2,11}, S_{3,7}, S_{4,8}, c_5^↑, S_{6,10}, S_{7,3}, S_{8,4}, d_9^↑, S_{10,6}, S_{11,2}, b_12^↓}_+

In[7]:= % == %%%

Out[7]= True
```

Let us give an indication of just how close the canonicalization of the free indices algorithm takes us towards a canonical configuration. We can do this by considering all possible equivalent configurations of a tensor product and seeing how many unique configurations remain after "canonicalizing the free indices".

```
In[8]:= signedConfig = ⟨R^{camn} R_a^{bd} R_{cnmbd}⟩_c;

In[9]:= equivalentConfigurations = generateConfigurations[signedConfig, S_T ⋃ S_X ⋃ S_D];

In[10]:= Union[canonicalizeFreeIndices[sc, S_T, S_X, S_D] &_sc /@ equivalentConfigurations]

Out[10]= {{S_{1,6}, S_{2,8}, S_{3,9}, S_{4,10}, S_{5,11}, S_{6,1}, S_{7,12}, S_{8,2}, S_{9,3}, S_{10,4}, S_{11,5}, S_{12,7}}_-,
          {S_{1,7}, S_{2,8}, S_{3,9}, S_{4,11}, S_{5,10}, S_{6,12}, S_{7,1}, S_{8,2}, S_{9,3}, S_{10,5}, S_{11,4}, S_{12,6}}_-,
          {S_{1,7}, S_{2,9}, S_{3,8}, S_{4,10}, S_{5,11}, S_{6,12}, S_{7,1}, S_{8,3}, S_{9,2}, S_{10,4}, S_{11,5}, S_{12,6}}_-}
```

We can see that we obtain only three configurations, whereas there was a total of 384 independent configurations before the canonicalization of the free indices.

```
In[11]:= Length @ equivalentConfigurations

Out[11]= 384
```

Of course if our free index canonicalizer were a perfect canonicalizer, then the last computation would have returned a single configuration. So, it is clear that canonicalizing the free indices alone does not always yield the canonical configuration. Nevertheless, it can be used to obtain good approximations to the canonical configuration.

Let us return to the issue of validation of the algorithm for the canonicalizing of the free indices of a signed configuration.

By examining the algorithm, it is evident that we split the free index canonicalization process into canonicalizing with respect to the degeneracy generators, and then continuing by canonicalizing with respect to the other generators. This is valid since the degeneracy generators swap blocks of indices and the indices inside separate blocks are never intermixed by the other permutations. Any free index that ends up in a specific "block" location after being canonicalized with respect to the degeneracy generators should remain in the same block independent of internal movement within the block due to the other generators. Thus, the

algorithm first shuffles the blocks of indices into their canonical positions and then proceeds to shuffle the free indices around inside each block.

Theorem 6.11.A: Given a signed configuration, a, and a JRDES set of generators, say $\mathcal{S}$ comprised of $\mathcal{S}_{\mathcal{T}}$, $\mathcal{S}_{\chi}$, and $\mathcal{S}_{\mathcal{D}}$, then `canonicalizeFreeIndices[`$a$, $\mathcal{S}_{\mathcal{T}}$, $\mathcal{S}_{\chi}$, $\mathcal{S}_{\mathcal{D}}$`]` returns a signed configuration equivalent to a in which the free indices of a are in their canonical positions.

Proof: Assume for a start that the algorithm terminates with a configuration where the lexicographically smallest free high index, say α, is in a position, say j, that is not its canonical position. In this case, the orbit of j under the group $\mathcal{G} = \langle \mathcal{S} \rangle$ contains a position, say i, where $i < j$; formally, $\exists\, i \in j^{\mathcal{G}}$ such that $i < j$. Therefore, it must be the case that one of the permutation generators in $\mathcal{S}$ will yield a smaller configuration, since the generator set $\mathcal{S}$ is JRDES. Consequently, we obtain a contradiction since termination of the algorithm means that a configuration was reached that could not be made smaller by any of the generators.

Since the generator set $\mathcal{S}$ contains all the generators for the extrema stabilizing subgroups, the above argument holds for the next smallest free high index; and so on. Obviously, an analogous argument holds for the free low indices. Thus, it is clear that if any free index were out of its canonical position, then one of the generators could move it closer to its canonical position. And any such movement would not disturb the canonical positions of smaller free indices, due to the definition of our ordering. ■

The above algorithm may seem quite wasteful of computational time in that we could have alternatively used a slightly different algorithm that successively moved each free index into a canonical position. This could be achieved by finding the generators that move the most significant free index, and then applying these generators to make the overall configuration smaller. Once this free index cannot be moved into a position that effects a smaller configuration, we stabilize the total set of generators by removing all generators that move this free index (which is now in its canonical position). We would then move the next free index into a canonical position, etc. However, the author has run several examples and found that the algorithm above, which canonicalizes in parallel, runs generally on the same scale of time as when canonicalizing sequentially.

After seeing how our algorithm maneuvers the free indices into their canonical positions, a natural question is raised. How come this works for free indices and yet we cannot apply the same technique to the dummy indices? After all, the algorithm makes no distinction concerning free versus summed indices: it just involves applying generators to the lowest configuration found so far, updating only when one finds a new lower configuration. Indeed, during the execution of our algorithm, we might well obtain a smaller configuration due to applying a permutation which moves just summed indices. For example

```
In[12]:= (3 ↔ 4)_ *c ⟨R a b b a⟩c <c ⟨R a b b a⟩c

Out[12]= True
```

After briefly reflecting on this, one should come to the realization that unlike free indices, dummy indices have no fixed "identity", hence no meaningful ordering. For example, after transforming a configuration with the index $s_{2,3}$, the result may well be a configuration in which $s_{2,3}$ does not appear.

In calculations that involve thousands of terms, it may sometimes be quicker simply to use the above process to get a good approximation to the canonical expression, thus probably greatly initially simplifying the calculation. Then, once many of the terms in the expression have coalesced, we could proceed to apply more exhaustive techniques.

6.11.2 Stabilized Subgroup Generators and Removal of Superfluous Generators

In the previous subsection we presented a simple algorithm to canonicalize the free indices of a given signed configuration. Once we have put the free indices into their canonical positions, we only need to use the generators of the stabilized subgroup of permutations which keep the free indices fixed. Since the free indices are the most important in our ordering, it follows that any comparison of configurations is first determined on the free indices and then on the summed indices. This means that if the free indices of a configuration $a \in C$ are in their canonical positions with respect to the group $\mathcal{G}$, then the minimum configuration of $\mathcal{G}_{stab} *_c a$ will be the same as the minimum configuration of $\mathcal{G} *_c a$ (where $\mathcal{G}_{stab}$ is the subgroup of $\mathcal{G}$ which does not move the free indices of the configuration a).

Because our generating set $\mathcal{S}$ is JRDES, we can generate $\mathcal{S}_{stab}$ simply by removing from $\mathcal{S}$ all generators that move one or more of the free indices of the configuration a, which has its free indices in canonical positions. Moreover, after stabilization, we can remove the superfluous complex generators from our generating set, since some of the complex generators may be expressible in terms of the degeneracy generators together with other complex generators. In fact, given complex generators ρ, $\rho' \in \mathcal{S}_\mathcal{X}$ for which ρ and ρ' are equivalent to each other under the degeneracy generators, that is, $\rho' = D^{-1}\,\rho\,D$ for D some product of degeneracy generators; then we can obviously remove either ρ or ρ' from our generating set. (It should be clear that ρ and ρ' are just shifted versions of each other.) For instance, if $\mathcal{S}_\mathcal{X} = \{\sigma, \pi\}$ where $\pi = \{3, 4, 1, 2, 5, 6, 7, 8\}$ and $\sigma = \{1, 2, 3, 4, 7, 8, 5, 6\}$, and $\mathcal{S}_\mathcal{D} = \{\rho\}$ where $\rho = \{5, 6, 7, 8, 1, 2, 3, 4\}$, then we can remove σ from $\mathcal{S}_\mathcal{X}$ since ρ and π are sufficient. That is because $\sigma = \rho \cdot \pi \cdot \rho^{-1}$, and so ρ and π generate the same set of configurations as ρ, π and σ. Of course, we will always need the full set of adjacent transpositions available after stabilization since they are needed in the transpositional canonicalization.

The code that implements the above reduction of the generating set and returns a minimal stabilized generating set is as follows.

```
In[13]:= stabilizePermutations[configuration_sign_, S𝒯 : _, S𝒳 : _, S𝒟 : _] :=
           Block[{stabilizedPositions = Cases[configuration, _↑n_ | _↓n_ → n],
                  stabilized𝒯, stabilized𝒳, stabilized𝒟},
             stabilized𝒯 = removePermutationsWhichMoveIndices[
```

```
        S_T, stabilizedPositions];
stabilizedD = removePermutationsWhichMoveIndices[
        S_D, stabilizedPositions];
stabilizedX = removePermutationsWhichMoveIndices[
        S_X, stabilizedPositions ⋃ Flatten[
                (Select[ρ, ρ_[m] < m &_m] &_ρ) /@ (σ_[2] &_σ) /@ stabilizedD]];
{stabilizedT, stabilizedX, stabilizedD}]
```

It simply finds the locations of all the canonicalized free indices and removes any generators that move these positions. For the slightly more complicated case of calculating the complex generators that should remain, we simply find all the larger indices that are moved by the degenerate name symmetries and remove all permutations in $\mathcal{S}_{\mathcal{X}}$ which move these indices.

The removal of permutations which move a given index can simply be performed by the following snippet of code.

```
In[14]:= removePermutationsWhichMoveIndices[S_List, indices_List] :=
           DeleteCases[S,
             (SignedPermutation[_, ρ_] /; (Or @@ (ρ_[m] ≠ m &_m) /@ indices)) |
               ((m_ /; (m ∈? indices)) ↔ _)_ | (_ ↔ (m_ /; (m ∈? indices)))_]
```

Let us give a small example of this behavior. Consider the symmetries of the following tensor product.

```
In[15]:= {S_T, S_X, S_D} = ⟨R^{αβστ} R^{ν}_{ατγ} R^{ξλ}_{ρβ} S^{μ}_{∈ξ}⟩_{S_T,X,D}

Out[15]= {{(1 ↔ 2)_, (3 ↔ 4)_, (5 ↔ 6)_, (7 ↔ 8)_,
            (9 ↔ 10)_, (11 ↔ 12)_, (13 ↔ 14)_+, (14 ↔ 15)_+},
          {{3, 4, 1, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}_+,
            {1, 2, 3, 4, 7, 8, 5, 6, 9, 10, 11, 12, 13, 14, 15}_+,
            {1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 9, 10, 13, 14, 15}_+},
          {{5, 6, 7, 8, 1, 2, 3, 4, 9, 10, 11, 12, 13, 14, 15}_+,
            {1, 2, 3, 4, 9, 10, 11, 12, 5, 6, 7, 8, 13, 14, 15}_+}}
```

Once we encode this product into a signed configuration and canonicalize the free indices in the resulting configuration, we can then calculate the generating set for the stabilized subgroup.

```
In[16]:= signedConfig = ⟨R^{αβετ} R^{ν}_{ατγ} R^{ξλ}_{ρβ} S^{μ}_{∈ξ}⟩_c

Out[16]= {S_{1,6}, S_{2,12}, S_{3,14}, S_{4,7}, ν↑_5, S_{6,1}, S_{7,4}, γ↓_8, S_{9,15}, λ↑_10, ρ↓_11, S_{12,2}, μ↑_13, S_{14,3}, S

In[17]:= newSignedConfig = canonicalizeFreeIndices[signedConfig, S_T, S_X, S_D]

Out[17]= {λ↑_1, S_{2,14}, S_{3,11}, ρ↓_4, ν↑_5, S_{6,12}, S_{7,10}, γ↓_8, S_{9,15}, S_{10,7}, S_{11,3}, S_{12,6}, μ↑_13, S_{14,2},

In[18]:= stabilizePermutations[newSignedConfig, S_T, S_X, S_D]

Out[18]= {{(9 ↔ 10)_, (11 ↔ 12)_, (14 ↔ 15)_+},
          {{1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 9, 10, 13, 14, 15}_+}, {}}
```

This generating set is, as expected, greatly reduced. Thus, far fewer configurations will have to be tested in order to determine the canonical configuration. We are now in a position to present the main optimized canonicalization algorithm.

6.11.3 The Optimized Canonicalization Algorithm

As previously stated, our basic canonicalization algorithm depended on three cornerstones: the permutation and relabeling action, the generation of configurations, and the ordering of configurations. For the optimized algorithm, we must add the canonicalization of the free indices and transpositional canonicalization. Here then are the formal steps in the optimized algorithm.

1. Expand all sums and products in the tensor expression.

For each tensor or tensor product in the sum, repeat steps 2 through 9.

2. Encode the tensor product into a signed configuration *signedConfiguration*.
3. Calculate the various subdivided generator sets corresponding to the symmetries of the tensor product.
4. Update *signedConfiguration* by moving the free indices of *signedConfiguration* into their canonical positions.
5. Calculate the generators for the stabilized subgroup which do not move the free indices of *signedConfiguration*; also remove any superfluous generators.
6. Relative to our stabilized generators, generate *allEquivalentConfigurations* — a set of signed configurations consisting of at least one minimum element from each $\mathcal{T}$-equivalence class of configurations equivalent to *signedConfiguration*.
7. Set *signedConfiguration* to a minimum configuration in *allEquivalentConfigurations*.
8. If *signedConfiguration* satisfies the `tensorIdenticallyZeroQ` test developed in §6.10 *Identically Zero Tensors*, then return zero
9. Else if the tensor is non-zero, reconstitute the canonical *signedConfiguration* into the canonical version of the original tensor product.

The code that accomplishes step 1 simply maps our `OptimizedCanonicalize` function over sums and factors out numbers from products (It is exactly the same as that of the `BasicCanonicalize` algorithm.)

```
In[19]:= OptimizedCanonicalize @ tensors_Plus := OptimizedCanonicalize /@ tensors
         OptimizedCanonicalize @ (num_ tensorProduct_) :=
            num OptimizedCanonicalize @ tensorProduct /; FreeQ[num, Tensor];
         OptimizedCanonicalize @ other_ = other;
```

Here is the implementation of the optimized algorithm, steps 2 through 9.

```
In[22]:= Symbolize[(S_T)_■];
```

```
In[23]:= OptimizedCanonicalize @ tensorProduct : (head_ @ __Tensor | _Tensor) :=
           Block[{S_T, S_X, S_D, (S_T)_•, productHead, tensorIdentifiers, signedConfiguration,
               allEquivalentConfigurations, tensorId, configurationLength},
             {productHead, tensorIdentifiers, signedConfiguration} =
               encodeTensors @ tensorProduct;
             configurationLength = signedConfiguration[[2]]_ten;
             {S_T, S_X, S_D, (S_T)_•} = calculateGenerators @ tensorIdentifiers;
             signedConfiguration =
               canonicalizeFreeIndices[signedConfiguration, S_T, S_X, S_D];
             {S_T, S_X, S_D} = stabilizePermutations[signedConfiguration, S_T, S_X, S_D];
             allEquivalentConfigurations =
               generateConfigurations_S_T[signedConfiguration, S_D ⋃ S_X];
             signedConfiguration = Min_c @ allEquivalentConfigurations;
             If[tensorIdenticallyZeroQ[
                 signedConfiguration, allEquivalentConfigurations, (S_T)_•], 0,
               reconstituteTensors[productHead, tensorIdentifiers, signedConfiguration]]]
```

In §6.10.5 *Zero Equivalence in the Optimized Algorithm*, we presented the code to determine if a tensor is identically zero (equivalent to zero from the symmetries alone) when we only have all the equivalent configurations up to transpositional canonicalization.

The one apparent remaining issue we have not tackled is the calculation of generators. The concepts behind `calculateGenerators` should now be familiar after covering §6.8 *Generators and Group Theoretic Underpinnings* and §6.7.2 *Specification of Generators*. We can superficially say that after obtaining the list of generators for each tensor, `calculateGenerators` proceeds to shift the generators along to where they occur in a configuration and then adds appropriate degeneracy generators according to the tensors in the tensor product. For a full discussion, see the code §D.5 *Constructing Generators from Primitive Generators* and in particular §D.5.2 *Calculate Generators*.

Unfortunately, for a complete working algorithm, we have not addressed one major item. Our algorithm may inadvertently raise or lower indices which we are not mathematically justified in doing. This annoying but easy to handle complication is the topic of the next section. However, let us first give a quick comparison of the basic and optimized canonicalization algorithms.

6.11.4 Optimized Canonicalization Examples

At this stage, it is enlightening to compare the timings and results of the basic canonicalization algorithm with that of the optimized canonicalization algorithm. Since the time taken to canonicalize expressions will fall below 0.1 seconds, we are starting to approach the limits of the recording-time granularity in *Mathematica*. Thus it is convenient to introduce a function which returns the *average* time taken over n successive runs. We denote this function, $\texttt{Timing}_n$.

```
In[24]:= Notation[
           Timing_n_ ⟹ Function[{expr}, AverageTiming[n_, expr], HoldAll]];
         SetAttributes[AverageTiming, HoldRest];
         AverageTiming[n_Integer /; n > 1, expr_] :=
           MapAt[# / n &, Timing[Do[expr;, {n - 1}]; expr], 1]
```

We can now use this function to increase the perceived accuracy of our timings. That is, the higher the n in $\texttt{Timing}_n$ the better the timing estimate.

Let us proceed to compare the execution speeds of the `BasicCanonicalize` and the `OptimizedCanonicalize` functions.

```
In[27]:= BasicCanonicalize[R^{ϵβγτ}_{τγ} R^{να}_{νξ} S^{μ}_{} ] // Timing_10

Out[27]= {2.37 Second, -R^{αγδζ}_{δζ} R^{βϵ}_{γξ} S^{μ}_{} }

In[28]:= OptimizedCanonicalize[R^{ϵβγτ}_{τγ} R^{να}_{νξ} S^{μ}_{} ] // Timing_100

Out[28]= {0.0418333 Second, -R^{αγδζ}_{δζ} R^{βϵ}_{γξ} S^{μ}_{} }

In[29]:= %%[[2]] == %[[2]]

Out[29]= True
```

Next, let us compare the speed at which the basic and the optimized algorithms recognize that something is zero.

```
In[30]:= BasicCanonicalize[R^{ab}_{cd} S^{e}_{abf} ] // Timing_50

Out[30]= {0.815 Second, 0}

In[31]:= OptimizedCanonicalize[R^{ab}_{cd} S^{e}_{abf} ] // Timing_100

Out[31]= {0.0226667 Second, 0}
```

To get an idea of the relative scaling with respect to complexity of the basic and the optimized algorithms, compare the previous timing results for canonicalizing a tensor product of three factors with the following results for a tensor product of four factors.

```
In[32]:= BasicCanonicalize[R^{αβγτ}_{ατγ} R^{ν}_{μβ} S^{ξλ}_{} S^{μ}_{ϵξ} ] // Timing

Out[32]= {94.4167 Second, R^{λαβγ}_{βδζη} R^{νδζη}_{αγϵ} S^{}_{} }

In[33]:= OptimizedCanonicalize[R^{αβγτ}_{ατγ} R^{ν}_{μβ} S^{ξλ}_{} S^{μ}_{ϵξ} ] // Timing_50

Out[33]= {0.065 Second, R^{λαβγ}_{βδζη} R^{νδζη}_{αγϵ} S^{}_{} }

In[34]:= %%[[2]] == %[[2]]
```

```
Out[34]= True
```

The next canonicalization does not finish for the basic algorithm, even with a 40 megabyte *Mathematica* kernel. However, the optimized algorithm handles it with ease.

```
In[35]:= OptimizedCanonicalize[R^{σβγτ ν}_{στγ} R^{ξλ}_{δβ} R^{εδ}_{αω} S^{η}_{ελ}] // Timing_50

Out[35]= {0.079 Second, R^{μ}_{λαω} R^{νβγδ} R^{ξεζλ} R_{βζγδ} S^{η}_{εμ}}
```

It is interesting to note that the time taken for the optimized canonicalization goes up dramatically when there are no free indices.

```
In[36]:= OptimizedCanonicalize[R^{σβγτ ν}_{στγ} R^{ξλ}_{δβ} R^{εα}_{νλ} S^{δ}_{εξ}] // Timing_10

Out[36]= {0.785 Second, R^{αβγδ} R^{εζηλ} R^{μ ν}_{γ ε} R_{δζηλ} S_{βμν}}
```

This increase in the time is due to there being many more configurations that have to be generated since there are fewer free indices to canonicalize and hence less restrictions on the generating set. However, the opposite is true for the basic algorithm since it generates *all* configurations and the presence of free indices greatly enlarges the set of configurations that have to be calculated. So as not to further cloud the issue with more specialized code, we will only report here that the memory requirements of the basic canonicalization algorithm are much higher than those of the optimized algorithm.

It is worthwhile pointing out that just canonicalizing the free indices and returning this result as something "close" to canonical form is typically about twice as fast as doing the full canonicalization. However, at times it can be up to an order of magnitude faster.

```
In[37]:= CanonicalizeBrief[R^{σβγτ ν}_{στγ} R^{ξλ}_{δβ} R^{εα}_{νλ} S^{δ}_{εξ}] // Timing_100

Out[37]= {0.0733333 Second, R^{αβγδ} R^{εζηλ} R^{μ ν}_{γ ε} R_{δζηλ} S_{βμν}}

In[38]:= %%[[2]] == %[[2]]

Out[38]= True
```

The downside of just canonicalizing the free indices is that we are unable to then answer the zero-equivalence question. The technique may, however, be useful in some specialized applications.

Before closing this section, for purposes of future execution speed comparisons, it is convenient to use the following test as one of our benchmarks.

```
In[39]:= OptimizedCanonicalize[R_{cd}^{mn} R^{abdc} R_{nmba}] // Timing_100

Out[39]= {0.107 Second, -R^{abcd} R^{ij}_{ab} R_{cdij}}
```

We will compare the execution speed of the optimized algorithm on the above tensor product with the execution speed after we make extensions for summed indices of fixed elevations, and then again when we make the further extension to class extended indices.

In conclusion, it is clear that the optimized algorithm is far superior to the basic algorithm. Let us now return to the aforementioned complication concerning the raising and lowering of indices.

6.12 Refinements for Partial Derivatives

6.12.1 Necessity of Indices with Fixed Elevations

Our algorithm, as so far developed, is computationally quite fast and applicable in many situations. Unfortunately we still have several issues to address if we are going to use this algorithm for computations in practice. Let us motivate the next extension we are about to provide. First, one of the assumptions we have made so far throughout the algorithm is that we can raise and lower the dummy index pairs to suit our algorithm. Unfortunately this is not always true. Consider:

$$\mathbf{T}_{i}\,\mathbf{S}^{i}{}_{,j} == \mathbf{T}_{i}\left(\mathbf{g}^{ik}\,\mathbf{S}_{k}\right)_{,j} == \mathbf{T}_{i}\,\mathbf{S}_{k,j}\,\mathbf{g}^{ik} + \mathbf{T}_{i}\,\mathbf{S}_{k}\,\mathbf{g}^{ik}{}_{,j} == \mathbf{T}^{k}\,\mathbf{S}_{k,j} + \mathbf{T}_{i}\,\mathbf{S}_{k}\,\mathbf{g}^{ik}{}_{,j} \tag{6.12.a}$$

This shows that when we raise and lower an index inside a partial derivative that is summed with an index outside the partial derivative, we must add a compensating term like $\mathbf{T}_{i}\,\mathbf{S}_{k}\,\mathbf{g}^{ik}{}_{,j}$. Consequently our algorithm, as it stands, will return incorrect answers since it does not include these extra terms in the canonical tensor. In geodesic coordinates — that is, when we have a locally flat metric — our results are still correct; however, we cannot always assume this is the case. The only solution is that for some summed indices, we must somehow remember whether they were initially lowered or raised and return them to their original elevations at the end of the algorithm.

An equation analogous to (6.12.a) is also true for covariant derivatives, except that the covariant derivative of the metric is always zero unless the system has a non-metric connection. In standard general relativity, all metrics are torsion free, so this is not a huge restriction. Furthermore, in many of our other more exotic applications, there is no problem with raising and lowering indices. Let us therefore proceed with the theory and modifications necessary to handle summed indices which must maintain a constant elevation.

Definition 6.12.A: An index with a *fixed elevation* is one that must maintain the same elevation, high or low, in the configuration throughout any manipulations. This includes dummy indices.

Obviously, in any given configuration, free indices already have a fixed elevation since they are of the form $\alpha^{\uparrow}_{pos}$ or $\alpha^{\downarrow}_{pos}$. However, our summed indices of the form $s_{i\,j}$ clearly only have the default elevation high if $i < j$ or low if $i > j$. Therefore, we must introduce an extension of the $s_{i\,j}$ with a fixed elevation.

```
In[1]:= Notation[s↑_i_,j_ ⟺ s[i_, j_, High]]
        Notation[s↓_i_,j_ ⟺ s[i_, j_, Low]]
```

The encoding routines in the code, §D.7.1 *Encoding Tensor Products into Configurations*, must of course take into account the fixed elevation of the necessary indices (although we will not comment further on how this is implemented). For instance, in $\mathbf{T}_{i}\,\mathbf{S}^{i}{}_{,j}$ the dummy index i must be fixed in both places that it occurs in the configuration.

```
In[3]:= ⟨T_i S^i_,j⟩_c
```

$$\text{Out[3]=}\quad \overline{\{s^{\downarrow}_{1,2},\ s^{\uparrow}_{2,1},\ j^{\downarrow}_{3}\}}_{+}$$

More generally, only indices which are involved in partial differentiation need ever maintain fixed elevations. For example, consider

```
In[4]:= ⟨R^abcd R^mn_ba,k R_dcnm⟩_c
```

$$\text{Out[4]=}\quad \overline{\{s^{\uparrow}_{1,8},\ s^{\uparrow}_{2,7},\ s_{3,11},\ s_{4,10},\ s^{\uparrow}_{5,13},\ s^{\uparrow}_{6,12},\ s^{\downarrow}_{7,2},\ s^{\downarrow}_{8,1},\ k^{\downarrow}_{9},\ s_{10,4},\ s_{11,3},\ s^{\downarrow}_{12,6},\ s^{\downarrow}_{13,5}\}}_{+}$$

The indices a, b, m, and n all appear inside a partial derivative so they cannot be raised or lowered at liberty. The other indices, c and d, do not have any constraint on their elevation, so are just normal $s_{i\,j}$ indices.

In the two examples so far, any summed index inside a partial derivative had its partner outside. If both indices of a summed pair occur inside a tensor, then we do not need to maintain any elevation on these dummy indices since they can be raised and lowered in conjunction. For instance, consider $\mathbf{R}^{i}{}_{aib,k}$. The index i does not need to be fixed since,

$$\begin{aligned}\mathbf{R}^{i}{}_{aib,k} &== \left(\mathbf{R}^{i}{}_{aib}\right)_{,k} == \\ &\left(\mathbf{g}^{ji}\,\mathbf{R}_{jaib}\right)_{,k} == \left(\mathbf{R}^{j}{}_{ja\,b}\right)_{,k} == \mathbf{R}^{j}{}_{ja\,b,k}\end{aligned} \tag{6.12.b}$$

Thus, dummy indices summed entirely within the scope of a partial derivative never need to have their elevations fixed because of that partial derivative. This is demonstrated when, for instance, we encode the product $\mathbf{R}^{i}{}_{aib,k}\,\mathbf{S}^{ak}$.

```
In[5]:= ⟨R^i_aib,k S^ak⟩_c
```

```
Out[5]= {s_{1,3}, s^{↓}_{2,6}, s_{3,1}, b^{↓}_{4}, s_{5,7}, s^{↑}_{6,2}, s_{7,5}}_+
```

In summary, both indices of a summed pair must have fixed elevations if and only if just one of the pair is inside a particular partial derivative.

Unfortunately, this opens up many issues all over again. What changes do we need to make to our basic permutation and relabeling action? What changes do we need to make to our ordering on signed configurations? Will we still generate all of the configurations using `generateConfigurations`? Will the canonicalization of free indices still work? What happens to transpositional canonicalization? Will our zero equivalence testing routines still work? In fact, almost everything we have developed so far must be subjected to a fresh appraisal.

Fortuitously, there are very few additions that must be made to accommodate indices with fixed elevations. In particular, we need only modify a single line in our ordering of configurations as well as make several additions to our transpositional canonicalization routines. These two modifications are covered in the following subsections. Everything else we have developed in the previous sections remains unmodified.

6.12.2 The Ordering of Configurations with Fixed Elevations

The modification to the ordering occurs through a change in the last line of our code that compares two signed configurations. Recall from §6.6.2 *Definition of the Ordering* that our code to compare two signed configurations is the following.

```
In[6]:= config1_{sign1_} ≶_c config2_{sign2_} :=
          Block[{ordering},
            ordering = Order[Sort @ Cases[config1, _High],
                  Sort @ Cases[config2, _High]];
            If[ordering == 0, ordering = Order[Reverse @ Sort @ Cases[config2, _Low],
                      Reverse @ Sort @ Cases[config1, _Low]];
              If[ordering == 0, ordering = Order[ config1 /. s → elevation,
                        config2 /. s → elevation];
                If[ordering == 0, ordering = Order[ config1, config2]]]];
            ordering]
```

We do not need to modify the first two lines containing `Order` statements since these just compare the two signed configurations according to their free indices, which remain unchanged by our extension. The third comparison just compares the relative elevations of the indices. Therefore, we need to add a second line to our code for `elevation`. In its entirety, the new code for `elevation` is the following.

```
In[7]:= elevation[i_Integer, j_Integer] := If[i < j, High, Low]
        elevation[_, _, elevation_] := elevation
```

Finally, the last comparison just compares the indices themselves, and thus we need to ignore any elevations of the summed indices. As a consequence, we can distill our comparison to just comparing the second position labels of corresponding indices. For example, say we are comparing $\mathbf{R}^{i\,j\;\;k}_{a\;\;,k}\mathbf{S}_{i\,j}$ with $\mathbf{R}^{i\,j\;\;k}_{a\;\;,k}\mathbf{S}_{j\,i}$. Let us examine their configurations.

```
In[9]:= ⟨R^{i j  k}_{a  ,k} S_{i j}⟩_c

Out[9]= {s^↑_{1,6}, s^↑_{2,7}, a^↓_3, s^↑_{4,5}, s^↓_{5,4}, s^↓_{6,1}, s^↓_{7,2}}_+

In[10]:= ⟨R^{i j  k}_{a  ,k} S_{j i}⟩_c

Out[10]= {s^↑_{1,7}, s^↑_{2,6}, a^↓_3, s^↑_{4,5}, s^↓_{5,4}, s^↓_{6,2}, s^↓_{7,1}}_+
```

It should be clear that our original code for comparing these two signed configurations will reach all the way to the last `Order` comparison, that is, to the line which has `Order[`*config*`1,`*config*`2]`. But when comparing indices with fixed elevation, say $s^{\uparrow}_{1,6}$ with $s^{\uparrow}_{1,7}$, we only need to compare 6 and 7; or in general, we only need to compare the second components of the `s[`*i,j,elevation*`]` or `s[`*i,j*`]` indices. Therefore, we change the fourth comparison to

```
Order[ config1 /. index_s :> index[[2]] , config2 /. index_s :> index[[2]] ].
```

The complete code is now the following.

```
In[11]:= config1_{-sign1_} ≲_c config2_{-sign2_} :=
           Block[{ordering},
             ordering = Order[Sort @ Cases[config1, _High],
                              Sort @ Cases[config2, _High]];
             If[ordering == 0, ordering = Order[Reverse @ Sort @ Cases[config2, _Low],
                                                Reverse @ Sort @ Cases[config1, _Low]];
               If[ordering == 0, ordering = Order[ config1 /. s → elevation,
                                                   config2 /. s → elevation];
                 If[ordering == 0, ordering = Order[ config1 /. index_s :> index[[2]],
                                                     config2 /. index_s :> index[[2]] ]]];
             ordering]
```

One important change that arises when indices with fixed elevations are included is that the minimum set of a set of signed configurations can now be potentially larger. Previously $mins_c(\mathcal{A})$ could contain at most two configurations, both having the same configuration but differing signs — see §6.6.5 *Minimum Configurations*. However, after introducing indices with fixed elevations and our new ordering on them, the set of minimums can be much larger. Let us quickly demonstrate this.

```
In[12]:= {s_{1,3}, s^↑_{2,4}, s_{3,1}, s^↓_{4,2}}_+ =_c {s^↑_{1,3}, s_{2,4}, s^↓_{3,1}, s_{4,2}}_+

Out[12]= True
```

It is clear that these configurations are considered the same with respect to our new ordering. This is consistent since, if say hypothetically the underlying tensor is T, and it has no covariant

or partial derivatives, then both configurations reconstitute to $\mathbf{T}^{a\,b}{}_{a\,b}$. Thus in effect, there is no substantial difference between the configurations. Actually, these configurations considered would not jointly arise in practice from the same tensor. However, the general principle holds: two signed configurations can be different but still *substantially* the same, that is $t_1 \neq t_2$ but $sign(t_1) = sign(t_2)$ and $t_1 =\!=_c t_2$.

There are only two further changes necessary to accommodate indices with fixed elevations. These modifications involve the transpositional canonicalization code as well as the determination of whether a tensor is identically zero. These are the topics of the next subsections.

6.12.3 Derivation of Refined Transpositional Canonicalization Criteria

This subsection details the necessary changes to the transpositional canonicalization operator in order to accommodate dummy indices with fixed elevations. It transpires that the only code that needs changing is that of `ReducingSwapQ`. Recall from §6.9.2 *ReducingSwapQ*, that $a \leftrightarrows_? b$ (or equivalently, `ReducingSwapQ[a,b]`) told us when we should swap the indices a and b in a configuration in order to effect a lower configuration. Let us first present the code for the necessary extensions to `ReducingSwapQ`, and then following this, let us prove that these extensions are the correct ones. That is, with these extensions, $a \leftrightarrows_? b$ will return true if and only if swapping a and b yields a smaller configuration. The extensions are as follows.

We should not swap $\uparrow \leftrightarrows \downarrow$, but we should swap $\downarrow \leftrightarrows \uparrow$.

```
In[13]:= s↑i_,m_ ⇆? s↓j_,n_ = False;
         s↓i_,m_ ⇆? s↑j_,n_ = True;
```

If both elevations are the same, then clearly we should just order using the normal conventions.

```
In[15]:= s↑i_,m_ ⇆? s↑j_,n_ = n < m;
         s↓i_,m_ ⇆? s↓j_,n_ = n < m;
```

And finally, for dealing with mixed kinds, we need to add the following four criteria.

```
In[17]:= s↓i_,m_ ⇆? sj_,n_ = m > n ∨ n > i;
         s↑i_,m_ ⇆? sj_,n_ = j < n < m;
         si_,m_ ⇆? s↓j_,n_ = n < m < i;
         si_,m_ ⇆? s↑j_,n_ = j > m ∨ m > n;
```

Technical Note: All of the above relations are subject to the constraint that `i < j`, which is always true in our uses of the reducing swap test.

Let us now prove that the above criteria are the "correct" ones. In §6.9.3 *Proof that Reducing Transpositions Yield Smaller Configurations*, we described the arguments leading to the formulation of the transpositional canonicalization operator. An essential thread running through these arguments is the use of mini-configurations. All possible mini-configurations

that could be reduced by the swapping of two generic summed indices $s_{i,m}$ and $s_{j,n}$ were found, and this led to a criterion as to when to swap these indices. In this revision we give the bare essentials of the code that deduces the new criteria when summed indices are allowed to have fixed elevations.

Theorem 6.12.A: Given any signed configuration $a \in C$, possibly containing dummy indices with fixed elevations, and a signed transposition $(i \leftrightarrow j)_{\pm}$ such that $i < j \leqslant a_{len}$ then $a_i \leftrightarrows_? a_j$ if and only if $(i \leftrightarrow j)_{\pm} *_c a \prec_c a$.

Proof: (Superficially sketched) The first part of the current proof is essentially the same as the first part of the proof of Theorem 6.9.C in §6.9.3 *Proof that Reducing Transpositions Yield Smaller Configurations*. Consequently, at this stage it may be helpful to re-examine the proof of Theorem 6.9.C. That proof arrived at a stage where we had to show that if $a_i = s_{i,m}$ and $a_j = s_{j,n}$ then $s_{i,m} \leftrightarrows_? s_{j,n}$ is true if and only if the transposed configuration a' where the i^{th}, j^{th}, m^{th} and n^{th} indices have become $s_{i,n}$, $s_{j,m}$, $s_{m,j}$ and $s_{n,i}$ is smaller that the configuration a where the i^{th}, j^{th}, m^{th} and n^{th} indices are $s_{i,m}$, $s_{j,n}$, $s_{m,i}$ and $s_{n,j}$.

The current proof diverges from the previous one at that stage of the proof just described above. We must now show our theorem to be true for $a_i = s_{i,m}$ or $s^{\uparrow}_{i,m}$ or $s^{\downarrow}_{i,m}$ together with $a_j = s_{j,n}$ or $s^{\uparrow}_{j,n}$ or $s^{\downarrow}_{j,n}$. For each of the 9 possible combinations, we must show that $a_i \leftrightarrows_? a_j$ is true if and only if the transposed configuration a', where the i^{th} and j^{th} indices have been swapped and relabeled, is smaller than the original configuration a.

For the code snippets that follow, it is convenient to introduce a unified notation for our $s_{i,j}$ indices.

```
In[21]:= Notation[ s^other____,j_ ⟺ s[i_, j_, other___] ]
```

As before, we define a function that gives relative values to a mini-configuration so that we can compare mini-configurations.

```
In[22]:= relativeValues [expr : {s_w_,_, s_x_,_, s_y_,_, s_z_,_}] :=
           expr /. {w → 1, x → 2, y → 3, z → 4}
```

Previously, transposing and relabeling the indices at positions i and j could be simply achieved by using the following replacements: $s_{i,m} \to s_{i,n}$, $s_{m,i} \to s_{m,j}$, $s_{j,n} \to s_{j,m}$, $s_{n,j} \to s_{n,i}$. Now however, due to the more varied possibilities for the i^{th} and j^{th} indices, we must achieve this by actually swapping the indices as follows.

```
In[23]:= transposeIandJ @ configuration_ :=
           Block[{
                 i = Position[configuration, s_i,_][[1,1]],
                 j = Position[configuration, s_j,_][[1,1]],
                 newConfiguration = configuration},
               {newConfiguration[[i]], newConfiguration[[j]]} =
                   {configuration[[j]], configuration[[i]]};
               newConfiguration /. {i → j, j → i}]
```

As before, if we transpose the i^{th} and j^{th} indices in the mini-configuration $\{s_{i,m}, s_{j,n}, s_{n,j}, s_{m,i}\}$ we get the mini-configuration $\{s_{i,n}, s_{j,m}, s_{n,i}, s_{m,j}\}$.

```
In[24]:= transposeIandJ @ {s_i,m, s_j,n, s_n,j, s_m,i}

Out[24]= {s_i,n, s_j,m, s_n,i, s_m,j}
```

But now, transposing the i^{th} and j^{th} indices works correctly even when summed indices of fixed elevation are involved.

```
In[25]:= transposeIandJ @ {s_i,n, s↑_j,m, s_n,i, s↓_m,j}

Out[25]= {s↑_i,m, s_j,n, s_n,j, s↓_m,i}
```

We can now determine if transposing and relabeling the indices at positions i and j in a mini-configuration will result in a smaller mini-configuration. (This code remains unchanged from that of §6.9.3 *Proof that Reducing Transpositions Yield Smaller Configurations.*)

```
In[26]:= transposedConfigurationIsSmallerQ @ reducedConfiguration_ :=
           Block[{transposedConfiguration = transposeIandJ @ reducedConfiguration},
               relativeValues[transposedConfiguration]+ <c
                   relativeValues[reducedConfiguration]+]
```

The following mini-configurations are made smaller by transposing the indices at positions `i` and `j`.

```
In[27]:= transposedConfigurationIsSmallerQ @ {s_i,m, s_j,n, s_n,j, s_m,i}

Out[27]= True

In[28]:= transposedConfigurationIsSmallerQ @ {s↑_i,m, s_j,n, s_n,j, s↓_m,i}

Out[28]= True
```

From each mini-configuration that can be made smaller by a transposition on i and j we obtain a condition on i, j, m and n.

```
In[29]:= relativeOrdering@ {s_w,_, s_x,_, s_y,_, s_z,_} := w < x < y < z
```

Thus, given a mini-configuration we can obtain a criteria on `i`, `j`, `m`, `n` for when we should swap the indices $s_{i,m}$ and $s_{j,n}$ (with or without various elevations). Basically, we take all permutations of the mini-configuration and select from these all

the mini-configurations which would be made smaller by swapping the indices at positions `i` and `j`, and relabeling. We further select only those mini-configurations for which `i` is less than `j`.

```
In[30]:= criterion @ configuration_ :=
           Or @@ relativeOrdering /@
                 Select[Permutations @ configuration,
                    transposedConfigurationIsSmallerQ[#] ∧
                          Position[#, s_{i,m}][[1, 1]] < Position[#, s_{j,n}][[1, 1]] &]
```

We can now just generate the criteria under which we should swap and relabel $s_{i,m}$ and $s_{j,n}$ (with or without various elevations). First, let us consider the cases when both indices to be swapped are of the same fixed elevation.

```
In[31]:= criterion @ {s↓_{i,m}, s↓_{j,n}, s↑_{n,j}, s↑_{m,i}}

Out[31]= i < j < n < m || i < n < j < m || i < n < m < j ||
           n < i < j < m || n < i < m < j || n < m < i < j

In[32]:= criterion @ {s↑_{i,m}, s↑_{j,n}, s↓_{n,j}, s↓_{m,i}}

Out[32]= i < j < n < m || i < n < j < m || i < n < m < j ||
           n < i < j < m || n < i < m < j || n < m < i < j
```

Both of these just reduce to $n < m \wedge i < j$; but since we are assuming $i < j$, this simplifies to $n < m$.

Next, consider the four cases when only one of the two indices to be swapped has a fixed elevation.

```
In[33]:= criterion @ {s↓_{i,m}, s_{j,n}, s_{n,j}, s↑_{m,i}}

Out[33]= i < j < n < m || i < j < m < n || i < n < j < m ||
           i < n < m < j || i < m < j < n || i < m < n < j || n < i < j < m ||
           n < i < m < j || n < m < i < j || m < i < j < n || m < i < n < j
```

This reduces to $\neg\,(m < n < i) \wedge i < j$; but since we are assuming $i < j$, this simplifies to $m > n \vee n > i$.

```
In[34]:= criterion @ {s_{i,m}, s↑_{j,n}, s↓_{n,j}, s_{m,i}}

Out[34]= i < j < n < m || i < n < j < m || i < n < m < j ||
           i < m < j < n || i < m < n < j || n < i < j < m || n < i < m < j ||
           n < m < i < j || m < i < j < n || m < i < n < j || m < n < i < j
```

This reduces to $\neg\,(j < m < n) \wedge i < j$; but since we are assuming $i < j$, this simplifies to $j > m \vee m > n$.

```
In[35]:= criterion @ {s_{i,m}, s↓_{j,n}, s↑_{n,j}, s_{m,i}}

Out[35]= n < m < i < j

In[36]:= criterion @ {s↑_{i,m}, s_{j,n}, s_{n,j}, s↓_{m,i}}

Out[36]= i < j < n < m
```

Finally, we consider the cases when the two indices to be swapped have opposite elevations.

```
In[37]:= criterion @ {s↑i,m, s↓j,n, s↑n,j, s↓m,i}

Out[37]= False

In[38]:= criterion @ {s↓i,m, s↑j,n, s↓n,j, s↑m,i}

Out[38]= i < j < n < m || i < j < m < n || i < n < j < m || i < n < m < j ||
           i < m < j < n || i < m < n < j || n < i < j < m || n < i < m < j ||
           n < m < i < j || m < i < j < n || m < i < n < j || m < n < i < j
```

These last two computations respectively return `False` and an expression equivalent to `True` under our assumptions.

Technical Note: Actually the last two computations respectively return $i > j$ and $i < j$. Only under our original assumption that $i < j$ do these criteria reduce to false and true respectively.

For comparison, the original criterion, where none of the indices has a fixed evaluation, is as follows.

```
In[39]:= criterion @ {si,m, sj,n, sn,j, sm,i}

Out[39]= i < j < n < m || i < m < j < n || i < m < n < j ||
           n < m < i < j || m < i < j < n || m < i < n < j
```

This criterion is exactly the same as the criterion previously derived in Theorem 6.9.C. Thus, we have shown that with the extensions to `ReducingSwapQ` presented previously in this subsection, $a_i \leftrightarrows_? a_j$ if and only if $(i \leftrightarrow j)_{\pm} *_c a \prec_c a$. ■

There is one last question that needs to be asked in regards to transpositional canonicalization. Will the algorithm to compute the transpositionally canonicalized configuration in a downhill fashion still work? Again, as in §6.9.4 *Transpositional Canonicalization and GenerateConfigurations*$_{\mathcal{T}}$, we must conjecture on the basis of strong evidence that it does.

Conjecture 6.12.A: Given a signed configuration $a \in C$, which can include indices with fixed elevations, and a set of adjacent transpositions $\mathcal{T}$ such that $\forall\, \tau \in \mathcal{T}\ a \preccurlyeq_c \tau *_c a$, then $\theta_{\mathcal{T}}(a) = a$.

Evidence: See the arguments in §C.2 *Evidence for Steepest Descent Conjecture*. ■

Introducing dummy indices with fixed elevations has affected the ordering of configurations; the encoding and reconstitution of configurations; the reducing swap criteria, which we have just handled; and the determination of whether a tensor is identically zero, which we next handle. To achieve closure on the issue of correctness, the reader might like to briefly skim all the results and code contained in this chapter up to this point, verifying that after the next subsection we will have considered all of the ramifications of introducing dummy indices with fixed elevations.

6.12.4 Determination of Identically Zero Tensors with Fixed Elevations

As mentioned earlier, the extension to configurations with indices of fixed elevation raises afresh all of the issues we have dealt with in earlier sections. Importantly, we now address the question of whether or not the ideas developed in §6.10 *Identically Zero Tensors* are still valid when dealing with configurations with fixed indices. In particular, does the algorithm presented in §6.10.5 *Zero Equivalence in the Optimized Algorithm* still function correctly?

It turns out that the ideas are still mostly valid except that we need slight extensions in a few specialized cases. Let us motivate these changes. Consider the following configurations.

```
In[40]:= config1 = {s[1,3], s^↑[2,4], s[3,1], s^↓[4,2]}_+
Out[40]= {s[1,3], s^↑[2,4], s[3,1], s^↓[4,2]}_+
In[41]:= config2 = {s^↑[1,3], s[2,4], s^↓[3,1], s[4,2]}_+
Out[41]= {s^↑[1,3], s[2,4], s^↓[3,1], s[4,2]}_+
```

Further, assume that the symmetries of our system are the following.

```
In[42]:= τ1 = (1 ↔ 2)_+; τ2 = (3 ↔ 4)_-;
         S_T = {τ1, τ2};
```

It should be noted that even with the extensions to transpositional canonicalization, these configurations are minimal.

```
In[44]:= Θ_{S_T}[config1]
Out[44]= {s[1,3], s^↑[2,4], s[3,1], s^↓[4,2]}_+
In[45]:= Θ_{S_T}[config2]
Out[45]= {s^↑[1,3], s[2,4], s^↓[3,1], s[4,2]}_+
```

In what follows, it is convenient to use the oppositely signed configuration to a given signed configuration.

```
In[46]:= oppositeConfiguration @ config_-sign_ = config_-sign;
         oppositeConfiguration @ 0 = 0;
```

Now, given a signed configuration a, does there exist a sequence of transpositions whereby we can transform a into $-a$? This was the fundamental question around which the results of §6.10.5 *Zero Equivalence in the Optimized Algorithm* were focused. Unfortunately, when indices with fixed elevations are involved, sometimes no such transpositions exist, even when the tensor under consideration is identically zero.

First, recall that when the summed indices do not have fixed elevations, such a sequence exists whenever the tensor represented by the configuration a is identically zero by the transpositional symmetries. A simple example of this is the following.

```
In[48]:= τ2 *c (τ1 *c {s1,3, s2,4, s3,1, s4,2}+ ) == {s1,3, s2,4, s3,1, s4,2}-

Out[48]= True
```

But when some dummy indices have fixed elevations, this is no longer true. The following configuration cannot be transformed into minus itself.

```
In[49]:= τ2 *c (τ1 *c config1 ) ≡ oppositeConfiguration @ config1

Out[49]= False
```

In fact, $\tau_2 *_c \tau_1 *_c$ *config*1 is instead equal to the opposite of *config*2.

```
In[50]:= τ2 *c (τ1 *c config1 ) == oppositeConfiguration @ config2

Out[50]= True
```

So, is the tensor corresponding to *config*1 identically zero or not? It is, since reconstituting indices in both *config*1 and *config*2 yield the same tensor. That is, say hypothetically the underlying tensor is T, and it has no covariant or partial derivatives; then both configurations reconstitute to $\mathbf{T}^{a\,b}{}_{a\,b}$. Another way to see that the tensor is identically zero is to note that *config*1 and the opposite of *config*2 have the same sign and according to our ordering function they are equal.

```
In[51]:= τ2 *c (τ1 *c config1 ) ==c config1

Out[51]= True
```

It is then natural to hypothesize the following generalization of our criteria. Loosely, to cover the $-a \sim_{\mathcal{T}} a$ case in our Theorem 6.10.D, is it enough that two configurations which are essentially equal (equal via $==_c$) are related via a pair of oppositely signed transpositions? This guarantees sufficiency but it unfortunately does not guarantee necessity. Consider the following

```
In[52]:= τ2 *c (τ1 *c {s1,3, s2,4↓, s3,1, s4,2↑}- )

Out[52]= {s1,3↓, s2,4, s3,1↑, s4,2}+
```

After transposing via τ_1 and τ_2, this configuration is not essentially equal to minus itself.

```
In[53]:= τ2 *c (τ1 *c {s1,3, s2,4↓, s3,1, s4,2↑}- ) ==c {s1,3, s2,4↓, s3,1, s4,2↑}-

Out[53]= False
```

Yet, since the $s_{i\,j}$ labels without fixed elevations can take on any elevation, we could reconstitute both configurations as the same thing. For instance, if the underlying tensor was say T, then we could reconstitute both these configurations as $\mathbf{T}_{a\,b}{}^{a\,b}$. Then, since the signs are opposite, we would have $\mathbf{T}_{a\,b}{}^{a\,b} == -\mathbf{T}_{a\,b}{}^{a\,b}$. Thus the tensor would be identically zero.

One might then be inclined to think that we can effectively ignore fixed elevations in signed configurations altogether. Unfortunately not. For instance, the following configuration has the same indices as the previous ones but different fixed elevations. The tensor represented by these configurations, however, is not identically zero.

```
In[54]:= τ2 *c (τ1 *c {s[1↑,3], s[2↓,4], s[3↓,1], s[4↑,2]}_)

Out[54]= {s[1↓,3], s[2↑,4], s[3↑,1], s[4↓,2]}+
```

Thus, the only time when our original criterion for determining the identically zero property does not work is when our transposition τ is operating on summed indices of fixed but opposite elevations. Formally, our generalization to Theorem 6.10.D is the following.

Theorem 6.12.B: Let $\mathcal{T}$ be a set of signed transpositions and let $a \in C$ be any signed configuration, possibly with indices of fixed elevations. Then $a =_c b$ and $-a \sim_{\mathcal{T}} b$ for some $b \in C$ if and only if there exists a signed transposition $\tau \in \mathcal{T}_\blacksquare$ such that either

(*i*) $\tau *_c a = -a$, or

(*ii*) both τ and $-\tau$ are present in $\mathcal{T}_\blacksquare$, or

(*iii*) $\tau \in \mathcal{T}_\blacksquare$ and $-\tau \in induced_a(\mathcal{T}_\blacksquare)$ and τ does not swap summed indices which are of opposite fixed elevation.

Proof: Essentially the same as the proof for the restricted version of the theorem, that is, Theorem 6.10.D. ■

Recall that Theorem 6.10.D relied upon Lemma 6.10.B. Similarly, Theorem 6.12.B, that is the revised version of Theorem 6.10.D, depends upon the following revised version of Lemma 6.10.B.

Lemma 6.12.A: Let τ be any signed transposition and $a \in C$ be any signed configuration, possibly with indices of fixed elevations. Then $\tau *_c a = -a$ if and only if $\tau = (m \leftrightarrow \overline{m})_-$ for some m such that a_m, the index at position m in the signed configuration a, is $s_{m,\overline{m}}$, and is *not* of fixed elevation.

Proof: Essentially the same as that of Lemma 6.10.B. ■

Due to the above Theorem 6.12.B above, we must modify our code. Here is the revised version of the code first presented in §6.10.5 *Zero Equivalence in the Optimized Algorithm*.

```
In[55]:= tensorIdenticallyZeroQ[
           minSignedConfiguration_, signedConfigs_List, T■ : _List] :=
         Block[
           {allT, minConfiguration = minSignedConfiguration[[2]], inducingRules},
           inducingRules = Flatten @ Cases[minConfiguration, s[i_,j_] → {i → j}];
           allT = T■ ⋃ (inducedTransposition /@ T■);
           (Union[sc[[2]] &sc /@ signedConfigs]ten < signedConfigsten) ⋁
             directlyZeroQ[minConfiguration, T■] ⋁
             zeroByComplementaryPairQ[minConfiguration, allT]]
```

`zeroByComplementaryPairQ` proceeds by removing the signs from the set $\mathcal{T}_{\blacksquare} \bigcup induced_a(\mathcal{T}_{\blacksquare})$ and finding any duplicates in the resulting set. If any of the index pairs in this set operate on indices which are not both fixed and of opposite elevations then the tensor is zero.

```
In[56]:= zeroByComplementaryPairQ[minConfiguration_, allT_] :=
           Or @@ (permissibleZeroIndicesQ @ minConfiguration[[#]] &) /@
                 Cases[Split @ Sort[sT[[2]] &sT /@ allT], {pair_, pair_} → pair]

In[57]:= permissibleZeroIndicesQ @ {s^↑_,_, s^↓_,_} = False;
         permissibleZeroIndicesQ @ {s^↓_,_, s^↑_,_} = False;
         permissibleZeroIndicesQ @ other_ = True;
```

We can demonstrate this code on the previous configurations used in this subsection. Since our only generators are transpositional the list of signed configurations consists only of the singleton signed configuration.

```
In[60]:= tensorIdenticallyZeroQ[config2, {config2}, S_T]

Out[60]= True

In[61]:= tensorIdenticallyZeroQ[config1, {config1}, S_T]

Out[61]= True
```

Finally, we can confirm that our code acts correctly on cases when the tensor would be identically zero but for the oppositely fixed elevations.

```
In[62]:= tensorIdenticallyZeroQ[
           Overscript[{s^↑_1,3, s^↓_2,4, s^↓_3,1, s^↑_4,2}, +], {Overscript[{s^↑_1,3, s^↓_2,4, s^↓_3,1, s^↑_4,2}, +]}, S_T]

Out[62]= False
```

There are other small differences that we have glossed over. For instance Theorem 6.9.A and its generalization Theorem 6.9.B, should really be amended to the following.

Theorem 6.12.C: $\theta_{\mathcal{T}}$ is a *semi-canonicalizing* function (relative to $\sim_{\mathcal{T}}$) on signed configurations corresponding to non-zero tensors. Formally, for any signed configurations a and b which are not $\mathcal{T}$-zero-equivalent we have

(*i*) $\theta_{\mathcal{T}}(a) \sim_{\mathcal{T}} a$

(*ii*) $b \sim_{\mathcal{T}} a \;\Rightarrow\; \theta_{\mathcal{T}}(b) =_c \theta_{\mathcal{T}}(a)$.

Proof: Essentially the same as the proof of Theorem 6.9.A. ■

Let us finally progress onto a brief comparison of the speeds of using fixed indices to that using standard indices.

6.12.5 Comparison of Dummy Indices: both Fixed and Standard

In this section we compare the canonicalized result of a tensor which has some indices of fixed elevation with a similar tensor that has no indices of fixed elevation. Our example tensor is chosen in such a way that the results are nearly identical, hence providing a fair comparison.

```
In[63]:= signedConfig = ⟨R^abcd R^mn _ba,k R_dcnm⟩_c
```

$$\text{Out[63]=}\ \{s^{\uparrow}_{1,8},\ s^{\uparrow}_{2,7},\ s_{3,11},\ s_{4,10},\ s^{\uparrow}_{5,13},\ s^{\uparrow}_{6,12},\ s^{\downarrow}_{7,2},\ s^{\downarrow}_{8,1},\ k^{\downarrow}_{9},\ s_{10,4},\ s_{11,3},\ s^{\downarrow}_{12,6},\ s^{\downarrow}_{13,5}\}_{+}$$

We next need the set of generators for this tensor product. Let us label this set $\mathcal{S}_{\mathcal{G}}$.

```
In[64]:= S_G = ⟨R^abcd R^mn _ba,k R_dcnm⟩_S
```

```
Out[64]= {(1 ↔ 2)_, (3 ↔ 4)_, (5 ↔ 6)_, (7 ↔ 8)_, (10 ↔ 11)_,
    (12 ↔ 13)_, {3, 4, 1, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13}_+,
    {1, 2, 3, 4, 7, 8, 5, 6, 9, 10, 11, 12, 13}_+,
    {1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 10, 11}_+}
```

Let us find the total number of configurations generated, how long it takes to generate them, and the overall minimum configuration.

```
In[65]:= generateConfigurations[signedConfig, S_G]_len // Timing
```

```
Out[65]= {0.35 Second, 64}
```

```
In[66]:= Min_c @ generateConfigurations[signedConfig, S_G]
```

$$\text{Out[66]=}\ \{s^{\uparrow}_{1,7},\ s^{\uparrow}_{2,8},\ s_{3,10},\ s_{4,11},\ s^{\uparrow}_{5,12},\ s^{\uparrow}_{6,13},\ s^{\downarrow}_{7,1},\ s^{\downarrow}_{8,2},\ k^{\downarrow}_{9},\ s_{10,3},\ s_{11,4},\ s^{\downarrow}_{12,5},\ s^{\downarrow}_{13,6}\}_{-}$$

For comparison, when we have no partial derivative, we obtain a very similar result.

```
In[67]:= signedConfig = ⟨R^abcd R^mn _ba R_dcnm⟩_c;
```

```
In[68]:= S_G = ⟨R^abcd R^mn _ba R_dcnm⟩_S;
```

```
In[69]:= generateConfigurations[signedConfig, S_G]_len // Timing
```

```
Out[69]= {0.4 Second, 64}
```

```
In[70]:= Min_c @ generateConfigurations[signedConfig, S_G]
```

$$\text{Out[70]=}\ \{s_{1,7},\ s_{2,8},\ s_{3,9},\ s_{4,10},\ s_{5,11},\ s_{6,12},\ s_{7,1},\ s_{8,2},\ s_{9,3},\ s_{10,4},\ s_{11,5},\ s_{12,6}\}_{-}$$

That is, but for the presence of the partial derivative, the resulting configurations are the same. It is perhaps easier to see this in the original tensor products rather than in the configurations.

Therefore, let us choose a tensor product that does not get reordered. Yet again, the presence of indices with fixed elevations does not unduly affect the resulting answer.

```
In[71]:= OptimizedCanonicalize[R^{abcd} R^{mn}_{ba} R_{dcnm}]

Out[71]= -R^{abcd} R^{ij}_{ab} R_{cdij}
```

```
In[72]:= OptimizedCanonicalize[R^{abcd} R^{mn}_{ba} R_{dcnm,k}]

Out[72]= -R^{abcd} R^{ij}_{ab} R_{cdij,k}
```

Finally, we should mention the typical execution speed penalty for including the extra facilities in our code to allow for the possibility that our configurations contain summed indices of fixed elevations. To illustrate this, we canonicalize the benchmark tensor product given in §6.11.4 *Optimized Canonicalization Examples*. In that subsection the timing for the following calculation took approximately 0.105 seconds.

```
In[73]:= OptimizedCanonicalize[R_{cd}^{mn} R^{abdc} R_{nmba}] // Timing_100

Out[73]= {0.1135 Second, -R^{abcd} R^{ij}_{ab} R_{cdij}}
```

Therefore, by allowing for summed indices which have fixed elevations, we have increased the overall execution time by something typically in the range of 5%. Thus we have been able to add a major extension to our algorithm at the expense of a extremely minor speed penalty.

6.13 Refinements for Mixed Index Classes

6.13.1 Desirability of Mixed Index Classes

Initially we presented the $s_{i,j}$ labels as if they were the "entire story". Then later, in §6.12 *Refinements for Partial Derivatives*, we conceded that actually this is not entirely true, and at certain stages we need to maintain fixed elevations for our summed indices. We must now introduce one final shift in our underlying structures for our canonicalization algorithm. The underlying format $s_{i,j}$ we have been using is not always viable in practice. At certain times, users might need to have dummy indices of different kinds in the same tensor. For instance, they might want space-time indices in some slots and just spatial indices in others. We must be able to handle canonicalizing such tensorial expressions. The tensor $\mathbf{R}^{\alpha\beta ij}\,\mathbf{T}_{ij}\,\mathbf{A}_{\alpha\beta}$ is a typical example of such a tensor with mixed class dummy indices. If such tensors are permissible in our calculations, then we need a way to canonicalize them.

Possibly the easiest and most direct solution is just to expand all such indices in the tensor under consideration. For example, expanding the aforementioned tensor would yield

$$\mathbf{R}^{\alpha\beta ij}\,\mathbf{T}_{ij}\,\mathbf{A}_{\alpha\beta} \longrightarrow \mathbf{R}^{00ij}\,\mathbf{T}_{ij}\,\mathbf{A}_{00} + \mathbf{R}^{0bij}\,\mathbf{T}_{ij}\,\mathbf{A}_{0b} + \mathbf{R}^{a0ij}\,\mathbf{T}_{ij}\,\mathbf{A}_{a0} + \mathbf{R}^{abij}\,\mathbf{T}_{ij}\,\mathbf{A}_{ab}$$

All of the resulting terms are canonicalizable with our existing algorithms since their dummy indices are all of the same kind. However, it would be nice to have the ability to handle such objects without always having to expand them. Moreover, in certain situations, it may not be possible to expand one kind of index into another kind. For example, we could be working with a barred set of indices which might represent indices in a different coordinate system. A specific instance of this is, say, the following.

$$\mathbf{R}^{a\bar{b}}{}_{c\bar{d}}\,\mathbf{S}_{\bar{b}a}$$

Of course the barred and unbarred indices are usually related by a coordinate transformation, so it may still be possible to expand the barred indices in terms of the unbarred indices. Overall though, just expanding the indices of one class in terms of indices of other classes is not always the ideal solution. Thus, it is desirable to adapt our canonicalization algorithm to handle such cases.

We must adapt our canonicalization algorithm in such a way as to somehow force each dummy index to "know" which class it belongs to. This "knowledge" must of course remain valid under the permutations of the indices in our configurations. Consequently, the obvious solution is to embed the index class information into each $s_{i,j}$ index. We now discuss two ways in which we could embed this information.

Let us briefly discuss a possible method we could use, but in the end opt not to use. We could incorporate the index class information into our summed indices by extending our summed index `s[i, j, elevation]` to `s[i, j, elevation, class]`, where *class* is a symbol naming an index class. In this way, all dummy indices will still be treated as real "dummy indices", but only amongst indices of the same class. For instance, in the tensor $\mathbf{R}^{\alpha\beta ij}\mathbf{T}_{ij}\mathbf{A}_{\alpha\beta}$ the indices α, β are space-time indices while the indices i, j are plain Cartesian spatial indices. The signed configuration for this tensor would be the same as that of $\mathbf{R}^{\gamma\nu da}\mathbf{T}_{da}\mathbf{A}_{\gamma\nu}$ but different from that of, say, $\mathbf{R}^{k\nu da}\mathbf{T}_{da}\mathbf{A}_{k\nu}$.

Unfortunately, if we want all index class names to be in the same place (i.e., the fourth position), we cannot just blithely tack on the class to the summed index since such a direct change would force all summed indices to be of fixed elevation. We would therefore need to introduce `Automatic` elevations, that is, summed indices of the form `s[i, j, Automatic, class]`. These would be indices of natural elevation but belonging to the index class *class*.

Similar to the above proposed dummy index extension, we instead adopt the following: we extend summed indices of the form `s[i, j]` to indices of the form `s[i, j, class]`, and similarly we extend indices of the form `s[i, j, elevation]` to indices of the form `s[i, j, class, elevation]`. Here *class* is a symbol naming an index class. However, when no confusion is possible, we will often refer to these symbols as just the 'index class', rather than the fuller 'name of an index class'. We must still be able to distinguish between the new `s[i, j, class]` and the old `s[i, j, elevation]`; but these two objects are of course easily distinguishable since any specific index class name we will use can never be `High` or `Low`, and these are the only permissible elevations.

Definition 6.13.A: A summed index with *index class inclusion*, or alternatively a *class extended index*, is an $s_{i,j}$, $s^{\uparrow}_{i,j}$, or $s^{\downarrow}_{i,j}$ index with an affixed index class. The class extended index must maintain the same class as the index changes position due to the action of permutations or symmetry operations.

Intuitively, due to the definition of our permutation and relabeling action, the class extended index must maintain the same class as the index changes position due to the action of symmetry operations. The following defines the notations for summed indices with an explicit index class affixed to the index.

```
In[1]:= Notation[ s_(i_, j_)^(class_)  <=> s[i_, j_, class_] ]
        Notation[ s_(i_, j_)^(class_, ↑)  <=> s[i_, j_, class_, High] ]
        Notation[ s_(i_, j_)^(class_, ↓)  <=> s[i_, j_, class_, Low] ]
        Notation[ s_(i_, j_)^(class_, elv_)  <=> s[i_, j_, class_, elv_] ]
```

Unfortunately, this latest extension to our dummy indices reopens many issues all over again. What changes do we need to our basic permutation and relabeling action? What changes do we need to make to our ordering on signed configurations? Will we still generate all of the configurations using `generateConfigurations`? Will the canonicalization of free indices still work? What happens to transpositional canonicalization? Will our zero equivalence testing routines still work? Yet again, almost everything we have developed so far must be subjected to another fresh appraisal.

As before, fortuitously, the necessary changes that must be made to accommodate our newly extended indices are relatively minor. In particular, we need only slightly modify our ordering of configurations in much the same way as we did previously in §6.12.2 *The Ordering of Configurations with Fixed Elevations*. Our transpositional canonicalization must be changed in its final stages, and our test to determine if a tensor is identically zero must also be changed. This is due to Theorem 6.10.D in general failing to hold under our extension. However, it remains true when the index classes of the dummy indices are the same. The following subsections develop the changes that are necessary to incorporate our extension.

6.13.2 The Inclusion of Index Classes in Summed Indices

This section introduces the notations for our extensions and gives several examples of signed configurations whose indices explicitly include index class symbols. An index class symbol represents the "class" or "kind" of an index. The concept of an index class, such as space-time, or Cartesian, etc., should be obvious. The two classes we will use in the present discussion are space-time indices and general indices. We will leave formal discussion of such things to §7.2.1 *Indices And Coordinates*. Henceforth, in this chapter, we will usually use "index class" to refer to an index class symbol.

As was the case for our extension to fixed elevations, the encoding routines in the implementation, §D.7.1 *Encoding Tensor Products into Configurations*, must take into account the inclusion of index classes into our summed indices. This capability is, of course, already incorporated into the code for the *Tensors* package. Index class inclusion can occur in a variety of ways. The main way it is specified is through options to our functions. Among other functions, the main canonicalization function `Canonicalize`, to be presented fully in §D.13 *The Complete Canonicalization Algorithm*, includes the capability to handle options. The relevant option for the purposes of index class inclusion is `IndexClassInclusion`. Let us set this option for the function `encodeTensors` and then examine various signed configurations arising from tensor products.

```
In[5]:= SetOptions[ encodeTensors, IndexClassInclusion → True]

Out[5]= {sortTensors → Automatic,
          IndexClassInclusion → True, MetricLocallyFlat → False}
```

Technical Note: Of course both `BasicCanonicalize` and our `OptimizedCanonicalize` could be modified in order to take options, but since they are largely pedagogical functions, we do not implement such an extension.

Let us examine the signed, classed, and labeled configuration corresponding to the tensor used in the motivating examples of the above subsection.

```
In[6]:= configuration = ⟨R^{α β i j} T_{i j} S_{α β}⟩_c
```

```
Out[6]= {s_{1,7}^{SpaceTimeIndices}, s_{2,8}^{SpaceTimeIndices}, s_{3,5}^{GeneralIndices}, s_{4,6}^{GeneralIndices}, s_{5,3}^{GeneralIndices}, s_{6,
```

This is somewhat awkward to read since the index classes are so long. Hence, let us introduce some further notation just for the index classes `SpaceTimeIndices` and `GeneralIndices`.

```
In[7]:= Notation[st_{i_,j_} ⟺ s[i_, j_, SpaceTimeIndices]]
        Notation[st^{↑}_{i_,j_} ⟺ s[i_, j_, SpaceTimeIndices, High]]
        Notation[st^{↓}_{i_,j_} ⟺ s[i_, j_, SpaceTimeIndices, Low]]
        Notation[sg_{i_,j_} ⟺ s[i_, j_, GeneralIndices]]
        Notation[sg^{↑}_{i_,j_} ⟺ s[i_, j_, GeneralIndices, High]]
        Notation[sg^{↓}_{i_,j_} ⟺ s[i_, j_, GeneralIndices, Low]]
```

Let us now re-examine the signed, classed, and labeled configuration.

```
In[13]:= configuration
```

```
Out[13]= {st_{1,7}, st_{2,8}, sg_{3,5}, sg_{4,6}, sg_{5,3}, sg_{6,4}, st_{7,1}, st_{8,2}}_+
```

We can operate on this configuration with signed permutations and transpositions just as we normally would.

```
In[14]:= {3, 4, 1, 2, 5, 6, 7, 8}_+ *_c configuration
```

```
Out[14]= {sg_{1,5}, sg_{2,6}, st_{3,7}, st_{4,8}, sg_{5,1}, sg_{6,2}, st_{7,3}, st_{8,4}}_+
```

```
In[15]:= (1 ↔ 2)_- *_c configuration
```

```
Out[15]= {st_{1,8}, st_{2,7}, sg_{3,5}, sg_{4,6}, sg_{5,3}, sg_{6,4}, st_{7,2}, st_{8,1}}_-
```

It is also illuminating to now examine the signed configuration of tensors where fixed elevations are required (due to the presence of partial derivatives).

```
In[16]:= ⟨R^{i}_{α i β, k} S^{α k}⟩_c
```

```
Out[16]= {sg_{1,3}, st^{↓}_{2,6}, sg_{3,1}, β^{↓}_{4}, sg_{5,7}, st^{↑}_{6,2}, sg_{7,5}}_+
```

Just as was the case previously in §6.12.1 *Necessity of Indices with Fixed Elevations* the indices that must maintain a fixed elevation have been "fixed". It is clear that our machinery for generating signed configurations from tensor products is functioning correctly with indices that explicitly include an index class.

By checking the definitions leading up to $*_c$ in §6.3 *Permutations and Configurations* and §6.4 *Labeling, Relabeling, and Group Actions,* one should convince oneself that there are no dependencies in these parts of the code on the fact that our $s_{i,j}$ have been extended. Since our fundamental action as well as the machinery surrounding our configurations remains largely

unchanged, it follows that the generation of configurations will also require no modification. However, once we re-examine the code for our ordering on signed configurations, $\prec_c$, we find that just as with the extension to fixed elevations, we need to make slight adjustments. This is the topic of the next subsection.

6.13.3 Modification to the Ordering for Class Extended Indices

The ordering on signed configurations, $\prec_c$, was originally defined in §6.6.2 *Definition of the Ordering*. Later, in §6.12.2 *The Ordering of Configurations with Fixed Elevations*, it was modified to incorporate handling of summed indices with fixed elevations. This change did not substantially alter the basic ordering. That is, we still ordered with respect to: (*i*) the free high indices, then (*ii*) the free low indices, then (*iii*) according to the elevations of the summed indices, and finally (*iv*) according to the order of the summed indices. This remained unchanged with the introduction of summed indices with fixed elevations. The only thing that changed was (*iii*): the elevations of some dummy indices — those having fixed elevations — were determined according to their fixed elevations rather than their natural elevations.

When including index classes into summed indices, we are introducing a new facet to our ordering. As we will see, index classes are different in nature from the other entries we have used so far in determining our ordering. We could, of course, radically alter our ordering and promote the index class to be of high priority in our ordering scheme. However, doing this would drastically alter most of the theory and some of the code developed so far. So we adopt the opposite approach. We suppress the influence of the index classes in our ordering scheme as much as is possible.

Thus, if after all the previous tests the configurations are indistinguishable and thus unorderable, then we try to order them according to the classes of each of the indices. To implement this, we just add a final line of code to our comparison. Here is the revised code.

```
In[17]:= config1_sign1_ ≲c config2_sign2_ :=
           Block[{ordering},
             ordering = Order[Sort @ Cases[config1, _High],
                    Sort @ Cases[config2, _High]];
             If[ordering == 0, ordering = Order[Reverse @ Sort @ Cases[config2, _Low],
                      Reverse @ Sort @ Cases[config1, _Low]];
               If[ordering == 0, ordering = Order[ config1 /. s → elevation,
                        config2 /. s → elevation];
                 If[ordering == 0, ordering = Order[ config1 /. index_s :→ index[[2]],
                          config2 /. index_s :→ index[[2]] ];
                   If[ordering == 0, ordering = Order[
                              config1 /. s → classOfSummedIndex,
                              config2 /. s → classOfSummedIndex]]]]];
             ordering]
```

The class of an index is determined by `classOfSummedIndex`. Any index without a specified class is treated as a "general" index.

```
In[18]:= classOfSummedIndex[_, _] := General;
         classOfSummedIndex[_, _, High | Low] = General;
         classOfSummedIndex[_, _, class_] = class;
         classOfSummedIndex[_, _, class_, _] = class;
```

We must also slightly revise our definition of the elevation of a summed index to take into account our introduction of summed indices with explicit index class inclusion.

```
In[22]:= elevation[i_Integer, j_Integer] := If[i < j, High, Low];
         elevation[_, _, _, elevation_] = elevation;
         elevation[_, _, elevation : (High | Low)] = elevation;
         elevation[i_Integer, j_Integer, class_] = If[i < j, High, Low];
```

In §6.6.3 *Examples of the Ordering*, we gave several examples of our ordering, including the following.

```
In[26]:= ⟨S^{abi j}_{i j}⟩_c ≺_c ⟨S^{aibj}_{ij}⟩_c ≺_c ⟨S^{aib j}_{i j}⟩_c
```

```
Out[26]= True
```

This ordering remains unchanged, as it should, when dummy indices from different classes are used.

```
In[27]:= ⟨S^{abi α}_{i α}⟩_c ≺_c ⟨S^{aβbj}_{βj}⟩_c ≺_c ⟨S^{aib j}_{i j}⟩_c
```

```
Out[27]= True
```

Only when the configurations are identical but for the index classes, does our extension distinguish between them.

```
In[28]:= ⟨R^{αβij} T_{ij} S_{αβ}⟩_c ≺_c ⟨R^{ijαβ} T_{αβ} S_{ij}⟩_c
```

```
Out[28]= False
```

The basic algorithm is now fully functional with respect to the inclusion of index classes in our summed indices.

```
In[29]:= BasicCanonicalize[R^{αβij} T_{ij} S_{αβ}]
```

```
Out[29]= 0
```

```
In[30]:= BasicCanonicalize[R^{abcd} R^{mn}_{ba} R_{dcnm,k}]
```

```
Out[30]= -R^{abcd} R^{ij}_{ab} R_{cdij,k}
```

```
In[31]:= BasicCanonicalize[R^{aβcδ} R^{mν}_{βa} R_{δcνm,k}]
```

```
Out[31]= -R^{aαbβ} R^{cγ}_{aα} R_{bβcγ,k}
```

These last two results are substantially the same. That is, up to index classes the results are identical. We now progress onto the necessary alterations to transpositional canonicalization.

6.13.4 Transpositional Canonicalization for Class Extended Indices I

In the previous subsection, we explicitly made the ordering of configurations have as little dependence on index class inclusion as possible. Thus, it might appear at first appraisal that transpositional canonicalization should have no dependence on index class inclusion. Indeed, when all the indices are of the same class it is obvious that we must default to exactly the same behavior as previously stated. Again recall from §6.9.2 *ReducingSwapQ* that $a \leftrightarrows_? b$, or equivalently `ReducingSwapQ[a,b]`, indicated when we should swap the indices a and b in the configuration in order to effect a lower configuration. Additional rules for `reducingSwapQ` were presented in §6.12.3 *Derivation of Refined Transpositional Canonicalization Criteria* for the extension to fixed elevations. This subsection presents the further rules needed to update `ReducingSwapQ` in order to allow the reducing swap operator to be used with class extended indices, as well as class extended fixed elevation indices.

The following rules are almost verbatim copies of the previous rules, however they have been extended by adding a slot for an index class. In these rules we ignore the index classes in our orderings. Only later do we talk about full transpositional canonicalization. Also since we either include index classes in all of our summed indices or in none of them, we never have to worry about `reducingSwapQ` working on one index that does have class inclusion and another that does not.

In[32]:= $s^{class1\_}_{i\_,m\_} \leftrightarrows_? s^{class2\_}_{j\_,n\_}$:= `compareS[i, j, m, n];`

Clearly we should not swap $\uparrow \leftrightarrows \downarrow$ but we should swap $\downarrow \leftrightarrows \uparrow$.

In[33]:= $s^{-,\uparrow}_{i\_,m\_} \leftrightarrows_? s^{-,\downarrow}_{j\_,n\_}$ = `False;`

$s^{-,\downarrow}_{i\_,m\_} \leftrightarrows_? s^{-,\uparrow}_{j\_,n\_}$ = `True;`

Also if both signs are the same then we should clearly just order using the normal conventions.

In[35]:= $s^{-,\uparrow}_{i\_,m\_} \leftrightarrows_? s^{-,\uparrow}_{j\_,n\_}$ = `n < m;`

$s^{-,\downarrow}_{i\_,m\_} \leftrightarrows_? s^{-,\downarrow}_{j\_,n\_}$ = `n < m;`

And finally when we mix kinds, the criteria is the following.

In[37]:= $s^{-,\downarrow}_{i\_,m\_} \leftrightarrows_? s^{-}_{j\_,n\_}$ = `m > n ∨ n > i;`

$s^{-,\uparrow}_{i\_,m\_} \leftrightarrows_? s^{-}_{j\_,n\_}$ = `j < n < m;`

$s^{-}_{i\_,m\_} \leftrightarrows_? s^{-,\downarrow}_{j\_,n\_}$ = `n < m < i;`

$s^{-}_{i\_,m\_} \leftrightarrows_? s^{-,\uparrow}_{j\_,n\_}$ = `j > m ∨ m > n;`

Let us now turn to the issue of transpositional canonicalization when our signed configurations involve class extended indices, with different classes. Since we are about to discuss slightly more theoretical issues surrounding transpositional canonicalization, let us formally refer to our new extended ordering as $\prec_{cn}$ (where *'cn'* is intended to suggest "*c*-new", even though in terms of code, we have just redefined $\prec_c$).

Assume in the following that we are starting with a class extended signed configuration, say $t \in C$. Recall from §6.6.5 *Minimum Configurations* that $mins_c(\langle \mathcal{T} \rangle *_c t)$, the set of minima of $\langle \mathcal{T} \rangle *_c t$, is defined as $\{a \in \langle \mathcal{T} \rangle *_c t \mid \forall\, b \in \langle \mathcal{T} \rangle *_c t,\ a \preccurlyeq_c b\}$. Also recall that this set of minima is intimately related to the definition of $\theta_{\mathcal{T}}$, that is $\theta_{\mathcal{T}}(t) \in mins_c(\langle \mathcal{T} \rangle *_c t)$ — see §6.9.1 *Transpositional Equivalence and Canonicalization*. With the extensions to the reducing swap operator given in this subsection, it must still be the case that $\Theta_{\mathcal{T}}(t) \in mins_c(\langle \mathcal{T} \rangle *_c t)$. This is due to the fact that all of the decisions / transpositions / permutations remain the same for our reducing swap operator since it ignores the index class. Thus Conjecture 6.9.A, and more generally Conjecture 6.12.A, must still be true.

However, unfortunately we are not guaranteed that $\Theta_{\mathcal{T}}(t)$ is minimum with respect to the new ordering. That is, it may be the case that $\Theta_{\mathcal{T}}(t) \notin mins_{cn}(\langle \mathcal{T} \rangle *_c t)$. Note the distinction between $mins_c$ and $mins_{cn}$: $mins_c$ is the minimum set with respect to the old order, $\prec_c$, and $mins_{cn}$ is the minimum set with respect to the new order, $\prec_{cn}$. When we were dealing with just standard configurations, the only possible difference between the elements of $mins_c(\mathcal{A})$ was that of sign. Now there can be a multitude of different configurations in $mins_c(\mathcal{A})$, all having the same basic layout, yet differing in any or all of (*i*) summed index classes, (*ii*) summed index elevations, or (*iii*) overall configuration sign. Let us illustrate these concepts with a simple configuration involving the Riemann tensor.

In[41]:=
$$\left\langle \mathbf{R}^{a\,\alpha}{}_{a\,\alpha} \right\rangle_c$$

Out[41]=
$$\overline{\{sg_{1,3},\ st_{2,4},\ sg_{3,1},\ st_{4,2}\}}_+$$

In[42]:=
$$\left\langle \mathbf{R}^{\alpha\,a}{}_{\alpha\,a} \right\rangle_c$$

Out[42]=
$$\overline{\{st_{1,3},\ sg_{2,4},\ st_{3,1},\ sg_{4,2}\}}_+$$

Under the old ordering, these were considered the same minimum configuration.

In[43]:=
```
SetOptions[ encodeTensors, IndexClassInclusion → False];
```
$$\left\langle \mathbf{R}^{\alpha\,a}{}_{\alpha\,a} \right\rangle_c =_c \left\langle \mathbf{R}^{a\,\alpha}{}_{a\,\alpha} \right\rangle_c$$

Out[44]=
```
True
```

Under the new ordering this is not so.

In[45]:=
```
SetOptions[ encodeTensors, IndexClassInclusion → True];
```
$$\left\langle \mathbf{R}^{a\,\alpha}{}_{a\,\alpha} \right\rangle_c \prec_c \left\langle \mathbf{R}^{\alpha\,a}{}_{\alpha\,a} \right\rangle_c$$

Out[46]=
```
True
```

Also, both of these configurations are in the same transpositional equivalence class.

```
In[47]:= ⟨R^{α a}_{α a}⟩_c == (3 ↔ 4)_ *_c ((1 ↔ 2)_ *_c ⟨R^{a α}_{a α}⟩_c)

Out[47]= True
```

Yet a standard downhill method will *never* reach the smaller configuration from the larger, since no single transposition will make the larger configuration smaller.

```
In[48]:= (1 ↔ 2)_ *_c ⟨R^{α a}_{α a}⟩_c ≺_c ⟨R^{α a}_{α a}⟩_c

Out[48]= False

In[49]:= (3 ↔ 4)_ *_c ⟨R^{α a}_{α a}⟩_c ≺_c ⟨R^{α a}_{α a}⟩_c

Out[49]= False
```

Thus, our original downhill approach is doomed to only find a configuration in $mins_c(\langle\mathcal{T}\rangle *_c t)$, which is not necessarily in $mins_{cn}(\langle\mathcal{T}\rangle *_c t)$. We need a new strategy. It turns out that we can create a new downhill strategy which will yield a configuration in $mins_{cn}(\langle\mathcal{T}\rangle *_c t)$. This is the topic of the next subsection.

6.13.5 Complementary-Effect Transposition Pairs

In the previous subsection we explained why just adapting the conventional transpositional canonicalization will lead us to a minimum in terms of the old ordering, but possibly not for the new ordering. In this section we will explain the approach we will take for our new downhill scheme that will complete the transpositional canonicalization.

We maintain the convention that $mins_c$ yields the set of minima with respect to the old order, $\prec_c$, and $mins_{cn}$ yields the set of minima with respect to the new order, $\prec_{cn}$. First, let us formalize the following concept which has already been used in the previous subsections, notably §6.10.4 *Theorems about Zero Equivalence in the Optimized Algorithm*.

Definition 6.13.B: Given a signed configuration $a \in C$, a *complementary-effect transposition pair* (for a) is a pair of signed transpositions $\tau_1, \tau_2 \in \mathcal{T}$ such that $unsigned(\tau_1) \neq unsigned(\tau_2)$ and $\tau_2 = \pm\, induced_a(\tau_1)$.

We will speak of *applying* a complementary-effect transposition pair, by which we will mean just applying one transposition after the other. The order in which we apply them is irrelevant since the transpositions must commute in the permutation group and thus, by the definition of a group action, must yield the same results in both cases. Given a signed configuration, say a, then applying a complementary effect transposition pair to a will yield a signed configuration a' having the same i,j labels on the summed indices. (This follows almost trivially from the

definition of an induced transposition, Definition 6.10.B, and the definition of our permutation and relabeling action $*_c$.) However, under a complementary-effect transposition pair, the index classes and / or the fixity of elevations may possibly move.

For instance, the transpositions $(1 \leftrightarrow 2)_+$ and $(3 \leftrightarrow 4)_+$ form a complementary-effect transposition pair relative to $\{s_{1,3},\ s^{\downarrow}_{2,4},\ s_{3,1},\ s^{\uparrow}_{4,2}\}_+$. Applying this pair results in a configuration with the same i, j labels but with a changed fixity of elevations.

```
In[50]:= (3 ↔ 4)+ *c ((1 ↔ 2)+ *c {s1,3, s2,4↓, s3,1, s4,2↑}+) == {s1,3↓, s2,4, s3,1↑, s4,2}+

Out[50]= True
```

We can think of the identically zero test presented in §6.10.5 *Zero Equivalence in the Optimized Algorithm* as searching for the special case of a complementary-effect transposition pair, τ_1 and τ_2, such that $\tau_2 *_c \tau_1 *_c a = a'$, where $a' =_c a$ and $sign(a') = -sign(a)$.

Theorem 6.13.A: Let $\mathcal{T}$ be a set of signed transpositions, and let $t \in C$ be a signed configuration with class extended indices, some of which are possibly of fixed elevations. Starting with a member of $mins_c(\langle\mathcal{T}\rangle *_c t)$ and successively applying all complementary-effect transposition pairs possible from $\mathcal{T}_\blacksquare$ which make the configuration smaller under the class extended ordering $\prec_{cn}$ yields a finite sequence terminating in a configuration lying in $mins_{cn}(\langle\mathcal{T}\rangle *_c t)$.

Proof: (Extremely superficially sketched) It is recommended that the reader review Lemma 6.10.C and Theorem 6.10.D. Say the starting configuration is $a \in mins_c(\langle\mathcal{T}\rangle *_c t)$, and also that one of the minimums with respect to $\prec_{cn}$ is $b \in mins_{cn}(\langle\mathcal{T}\rangle *_c t)$. By the arguments similar to those previously given in §6.6.5 *Minimum Configurations*, we note that $mins_{cn}(\langle\mathcal{T}\rangle *_c t)$ can contain at most two signed configurations — that is, the same configuration appearing with both signs. So ignoring signs, we can speak of the minimum b. Since $b \sim_{\mathcal{T}} a$ we must be able to find $\tau_i \in \mathcal{T}$ such that $b = \prod_{i=1}^{k} \tau_i *_c a$. If a is equal to b, then we are done. Thus assume $a \neq b$. Consider the lowest index m such that $a_m \neq b_m$. Clearly, a_m must be a summed index since the elements of $mins_c(\langle\mathcal{T}\rangle *_c t)$ only differ on their index classes, and only summed indices have index classes. Thus, $a_m = s^{cfass}_{m,\overline{m}}$. Also, again since the elements of $mins_c(\langle\mathcal{T}\rangle *_c t)$ only differ on their index classes, it must be the case that $b_m = s^{cfass'}_{m,\overline{m}}$. (We are ignoring elevations at this stage.)

By similar arguments as those given in Theorem 6.10.D, we can find $\tau_i'' \in \mathcal{T}_\blacksquare$ such that $\prod_{i=1}^{k} \tau_i = (\prod_{i=1}^{k''} \tau_i'') \cdot (\overline{m} \leftrightarrow \overline{n})_\pm \cdot (m \leftrightarrow n)_\pm$ where the τ_i'' do not move m or $\overline{m}$. In fact, it can be shown that we can find τ_i'' such that the smallest position moved is greater than m. (If there were a smaller position moved by the τ_i'' then we could propagate this transposition through to the right-hand side, as was done in Lemma 6.10.C, thus either contradicting the fact that a_m was the first index in a differing from those of b or resulting in an elimination of the transposition.) Call $a' = (\overline{m} \leftrightarrow \overline{n})_\pm \cdot (m \leftrightarrow n)_\pm *_c a$. Since $(\overline{m} \leftrightarrow \overline{n})_\pm$ and $(m \leftrightarrow n)_\pm$ form a complementary-effect transposition pair, it must still be the case that $a' \in mins_c(\langle\mathcal{T}\rangle *_c t)$. (With fixed elevations, the explanation is more involved, but it is still true.) Furthermore, since none of the transpositions τ_i'' can move the m^{th} index, it must be the case that

$a'_m = b_m$. But since no position smaller than m was moved, a' and b must still also agree on all positions smaller than m. Collectively then, $a'_i = b_i$ for all $i \leqslant m$. That is, we have moved the index class difference further along the configuration. Formally, $a' \prec_{cn} a$. By successively shifting the index class difference further and further right, we must eventually arrive at the overall minimum signed configuration b. (The inclusion of indices with fixed elevations does not substantially alter the above arguments.)

Thus, collectively, we know that if we are not at the smallest overall configuration, we can always find a complementary-effect transposition pair which will produce a smaller one. Since we are continually finding smaller and smaller configurations, we must arrive at an overall minimum signed configuration. ■

Actually, on the basis of personal observation and experimentation, we conjecture that when our $\langle \mathcal{T} \rangle$ is adjacently generated, it is possible to just use $\mathcal{T}$ in Theorem 6.13.A, as opposed to $\mathcal{T}_\blacksquare$.

Conjecture 6.13.A: Let $\mathcal{T}$ be a set of *adjacent* signed transpositions, and let $t \in C$ be a signed configuration with class extended indices, some of which are possibly of fixed elevations. Starting with a member of $mins_c(\langle \mathcal{T} \rangle *_c t)$ and successively applying all complementary-effect transposition pairs possible from $\mathcal{T}$ which make the configuration smaller under the class extended ordering $\prec_{cn}$ yields a finite sequence terminating in a configuration lying in $mins_{cn}(\langle \mathcal{T} \rangle *_c t)$.

Evidence: If we have a transposition on $(m \leftrightarrow n)$ then we will have transpositions on all the positions in between since $\mathcal{T}$ consists of adjacent transpositions. On a strictly intuitive level, if we have a double transposition on a $\prec_c$-minimum configuration, then the indices will be so ordered that there exists intermediate indices which we should shuffle instead. This parallels, say, the situation when we order {w,x,y,y,z} in lexically descending order. It is true that we could shuffle z with one of the x's, but since there must be intermediate indices, we could shuffle with these instead. This is intuitively akin to the bubble sort method for sorting a list.

Besides the intuitive notions above, there exists strong concrete evidence that the conjecture is true. By brute force we can test all signed configurations with all possible class extensions and all possible elevations on n indices with all possible adjacent transposition generating sets. We perform an indirect verification of our conjecture in §C.2 *Evidence for Steepest Descent Conjecture* completely for $n = 6$. Unfortunately, because of combinatorial explosion, we must limit the testing of larger configurations to random sampling methods. If we ignore configurations with fixed elevations, then the number of unique configurations on 8 indices drops to a level where we can verify the conjecture. ■

We can combine Conjecture 6.13.A with Conjecture 6.12.A to arrive at the following conjecture.

Conjecture 6.13.B: Given a signed configuration $a \in C$, which can include indices with fixed elevations and/or index class extensions, and a set of adjacent transpositions $\mathcal{T}$ such that:

(*i*) $\forall\, \tau \in \mathcal{T}\ a \preccurlyeq_c \tau *_c a$, and

(*ii*) $\forall\, \tau, \overline{\tau} \in \mathcal{T}\ a \preccurlyeq_c \overline{\tau} *_c \tau *_c a$ (where τ, $\overline{\tau}$ are complementary-effect transposition pairs),

then $\theta_{\mathcal{T}}(a) = a$.

Evidence: See the arguments in §C.2 *Evidence for Steepest Descent Conjecture.* ■

The next subsection presents the modifications we must make in the implementation of our transpositional canonicalization operator.

6.13.6 Transpositional Canonicalization for Class Extended Indices II

This subsection presents the actual code that implements our extension to transpositional canonicalization. To briefly recap, our original transpositional canonicalization searches through all transpositions, performing only those for which the original reducing swap criterion is true. This downhill method yields a configuration in $mins_c(\langle\mathcal{T}\rangle *_c t)$. Our modification will be the addition of further searching through all the transpositions looking for complementary-effect transposition pairs that will further reduce the configuration. When we can find no more such transposition pairs we have arrived at a configuration in $mins_{cn}(\langle\mathcal{T}\rangle *_c t)$.

Here is the modified code for the transpositional canonicalization. It is very similar to the code given in §6.9.4 *Transpositional Canonicalization and GenerateConfigurations*$_{\mathcal{T}}$.

```
In[51]:= Θ_S_List[ configuration_signC_ ] :=
           Block[ {newConfiguration = configuration,
                   newSign = signC, partialConfig = {}, 𝓕, N, 𝒯 = S},
             While[partialConfig ≠ newConfiguration,
               partialConfig = newConfiguration;
               transposeIfMoreCanonical /@ 𝒯];

             complementaryTransposeIfMoreCanonical /@
               Cases[𝒯, (i_ ↔ j_)_signt_ /; mixesIndexClassesQ@ partialConfig⟦{i, j}⟧];

             While[partialConfig ≠ newConfiguration,
               partialConfig = newConfiguration;
               complementaryTransposeIfMoreCanonical /@ 𝒯];

             partialConfig_newSign]
```

One of the changes in our code is that we partially unwind our second closure loop by first explicitly looking for cases when we would actually perform a swap. In many cases, we never have to do any complementary double swaps at all. Thus, unwinding the first step is done for the sole purpose of execution speed.

```
In[52]:= mixesIndexClassesQ @ {s_,_^class1_?ValidIndexClassQ, ___, s_,_^class2_?ValidIndexClassQ, ___} :=
           class1 ≠ class2
         mixesIndexClassesQ @ other_ = False;
```

We must now determine if a given transposition can be one half of a complementary-effect transposition pair which will reduce the current configuration. We therefore pass the indices which this transposition would operate on along with the sign of the operation, to the auxiliary function `reducingComplementarySwap`.

```
In[54]:= complementaryTransposeIfMoreCanonical @ (i_ ↔ j_)_sign_ :=
           reducingComplementarySwap[sign, newConfiguration_[[{i, j}]]]
```

Without loss of generality, we can assume we are operating on a pair of indices (i,j) that are further to the left than their complementary pair $(\bar{i},\bar{j})$, since if the pair (i,j) is further to the right, then the complementary transposition will operate on the complementary pair $(\bar{i},\bar{j})$. Also, the whole purpose of performing the complementary double transposition was to achieve a lower configuration, so the index class ordering must be improved by the swap. In addition to all of these restrictions, since the transpositions are adjacent, we know that the first index of the complementary pair $(min(\bar{i}, \bar{j}))$ must be further right then either index of the pair (i,j). These considerations, in part, collectively lead us to the following code.

```
In[55]:= reducingComplementarySwap[sign_,
             {ai : s_i_,m_^class1_?ValidIndexClassQ, ___, aj : s_j_,n_^class2_?ValidIndexClassQ, ___}] /;
           class2 <_lex class1 ∧ i < j < Min[m, n] ∧
             doubleSwapIsMoreCanonicalQ[ai, aj] :=
         Block[{foundSign},
           foundSign = Cases[T, (Min[m, n] ↔ Max[m, n])_sign_ → sign];
           If[foundSign ≠ {},
             doubleSwapThem[sign (First @ foundSign), i, j, m, n];]]

In[56]:= reducingComplementarySwap @ other___ = False;
```

It turns out that just by examining the two indices to be swapped, `doubleSwapIsMoreCanonicalQ` can determine whether we should perform the double swap if the complementary transposition exists. (In a sense it is the counterpart to `reducingSwapQ`.) By the way in which `doubleSwapIsMoreCanonicalQ` is called, it is the case in the following that *class1* is always lexically greater than *class2*. (Recall that under our ordering $<_{cn}$ we want the lexically smaller classes to be as far left as possible.)

```
In[57]:= doubleSwapIsMoreCanonicalQ[s_i_,m_^class1_, s_j_,n_^class2_] = True;

In[58]:= doubleSwapIsMoreCanonicalQ[s_i_,m_^class1_,↓, s_j_,n_^class2_,↓] = True;
         doubleSwapIsMoreCanonicalQ[s_i_,m_^class1_,↑, s_j_,n_^class2_,↑] = True;

In[60]:= doubleSwapIsMoreCanonicalQ[s_i_,m_^class1_,↑, s_j_,n_^class2_] = True;
         doubleSwapIsMoreCanonicalQ[s_i_,m_^class1_, s_j_,n_^class2_,↑] = True;
```

All other cases would either never occur in practice, or would not result in a lower configuration.

```
In[62]:= doubleSwapIsMoreCanonicalQ[_, _] = False;
```

For instance, the conceivable case $s^{class1\_,\downarrow}_{i\_,m\_}$, $s^{class2\_}_{j\_,n\_}$ would never occur since it can be made smaller by the transposition $(i \leftrightarrow j)$. Also for instance, the case $s^{class1\_,\uparrow}_{i\_,m\_}$, $s^{class2\_,\downarrow}_{j\_,n\_}$ may occur in practice, but we would be altering the elevations by performing the $(i \leftrightarrow j)$ swap, and so the double transposition would definitely not make the configuration smaller.

To convince oneself that the choices made in `doubleSwapIsMoreCanonicalQ` will lead to smaller configurations, we could easily generalize the arguments we made using mini-configurations — see Theorem 6.9.C in §6.9.3 *Proof that Reducing Transpositions Yield Smaller Configurations* and Theorem 6.12.A in §6.12.3 *Derivation of Refined Transpositional Canonicalization Criteria.*

Once we have determined that the double transposition would be beneficial, we proceed to determine if the complementary transposition exists. If such a transposition actually exists, then we have to carry out the actual swap. This can be done in a manner very similar to that given in §6.9.4 *Transpositional Canonicalization and GenerateConfigurations*$_T$.

```
In[63]:= doubleSwapThem[signττ_, i_, j_, m_, n_] := (
           {newConfiguration⟦i⟧, newConfiguration⟦j⟧, newConfiguration⟦m⟧, newConfiguration⟦n⟧} =
             newConfiguration⟦{j, i, n, m}⟧;
           newSign = newSign signττ;
           newConfiguration =
             Replace[newConfiguration, {i → j, j → i, n → m, m → n}, {2}];);
```

Let us demonstrate this new extension in practice. Consider the following class extended signed configurations with some of the indices having fixed elevations.

```
In[64]:= config1 = {st_{1,3}, sg^↑_{2,4}, st_{3,1}, sg^↓_{4,2}}_+ ;
         config2 = {st^↑_{1,3}, sg_{2,4}, st^↓_{3,1}, sg_{4,2}}_+ ;
```

Further, assume that the symmetries of our system are the following.

```
In[66]:= τ1 = (1 ↔ 2)_+ ; τ2 = (3 ↔ 4)_- ;
         S_T = {τ1, τ2};
```

Under the old transpositional canonicalization, both of these configurations were minimal. With our new transpositional canonicalization, we can see that they can both be made smaller by our transpositional symmetries.

```
In[68]:= Θ_{S_T}[config1]

Out[68]= {sg^↑_{1,3}, st_{2,4}, sg^↓_{3,1}, st_{4,2}}_-

In[69]:= Θ_{S_T}[config2]

Out[69]= {sg_{1,3}, st^↑_{2,4}, sg_{3,1}, st^↓_{4,2}}_-
```

Let us address the last remaining issue, that of identically zero tensors when we include index classes in our configurations.

6.13.7 Identically Zero Tensors and Class Extended Indices

Let us first motivate the extensions which we will make to our code to determine whether or not a tensor is identically zero. Consider the following basic canonicalization of a tensor.

```
In[70]:= BasicCanonicalize[R^{α i β j} S_{i α}]

Out[70]= R^{j β a α} S_{a α}
```

One might briefly ponder why the last answer was not zero. After all, R is anti-symmetric on α and i, and it is multiplied by S which is symmetric on α and i. Why is this tensor not zero? The answer is that, in general, symmetric times anti-symmetric must be working on the same class of indices, for it fundamentally depends on being able to substitute new dummy labels for old labels.

```
In[71]:= BasicCanonicalize[R^{α i β j} S_{i α}] == - BasicCanonicalize[R^{i α β j} S_{α i}]

Out[71]= True
```

We have already modified the code for determining whether a tensor is identically zero in order to accommodate fixed elevation indices. In what follows, it is strongly recommended that the reader review that subsection, that is, §6.12.4 *Determination of Identically Zero Tensors with Fixed Elevations*. In that subsection, the code for `zeroByComplementaryPairQ` was given. This routine searches the set $\mathcal{T}_{\blacksquare}$ for a complementary-effect transposition pair such that the minimum configuration is taken to an equivalent of minus itself. Recall from §6.12.4 that even if we found a complementary-effect transposition pair with overall minus sign, we were still not guaranteed that the tensor was zero. To ascertain whether the complementary pair is operating on the right kind of indices, we introduced the testing function `permissibleZeroIndicesQ`. Now we must make a further adjustment to this code in order to incorporate class extended indices.

As in §6.12.4, if we are operating on class extended indices with fixed opposite elevations, then we cannot effect a zero tensor by a complementary pair of transpositions.

```
In[72]:= permissibleZeroIndicesQ @ {s_{_,_}^{_,↑}, s_{_,_}^{_,↓}} = False;
         permissibleZeroIndicesQ @ {s_{_,_}^{_,↓}, s_{_,_}^{_,↑}} = False;
```

The additional condition we must add is that when we are operating on class extended indices with different classes, then we again cannot obtain a zero tensor by a complementary pair of transpositions.

```
In[74]:= permissibleZeroIndicesQ @
           {s_{_,_}^{class1_?ValidIndexClassQ,___}, s_{_,_}^{class2_?ValidIndexClassQ,___}} := class1 ≡ class2;
```

In all other cases, it remains true that an oppositely signed complementary pair of transpositions will still yield a zero tensor.

Finally, we need to change slightly one of the tests for `directlyZeroQ`. As long as we are not operating on a pair of indices with fixed elevations, then a tensor can still be directly zero.

```
In[75]:= Clear @ reduceIndex

In[76]:= reduceIndex @ s_[i_, j_] := s @ Min[i, j];
         reduceIndex @ s_[i_?ValidIndexClassQ, j_] := s @ Min[i, j];
         reduceIndex @ h_[index_, ___] = h @ index;
```

We are now ready to observe the combined changes in this section and give several examples comparing the speed of our extended algorithm to that of our original algorithm.

6.13.8 Speed Penalties due to Class Extended Indices

We have now presented the theoretical ideas and actual code necessary to incorporate class extended indices into our optimized canonicalization algorithms. This subsection briefly examines the influence of these changes to the execution speed of the algorithm.

Unfortunately, including index classes in the summed indices of our configurations will lead to a slow down. Let us use our benchmark tensor product, introduced in §6.11.4 *Optimized Canonicalization Examples*, to evaluate the relative performance of our latest extension.

```
In[79]:= OptimizedCanonicalize[R_{cd}^{mn} R^{abdc} R_{nmba}] // Timing_100

Out[79]= {0.145833 Second, -R^{abcd} R^{ij}_{ab} R_{cdij}}
```

This particular calculation took approximately 0.110 seconds in §6.12.5 *Comparison of Dummy Indices : both Fixed and Standard*. Thus, the further slowdown in our algorithm for this benchmark tensor product is close to 30%. Relative to our original optimized algorithm without any extensions, this is a slowdown of approximately 35%. Let us compare the other larger calculation which we embarked upon earlier in §6.11.4.

```
In[80]:= OptimizedCanonicalize[R^{σβγτ} R^{ν}_{στγ} R^{ξλ}_{δβ} R^{εα}_{νλ} S^{δ}_{εξ}] // Timing_10

Out[80]= {1.04667 Second, R^{αβγδ} R^{εξηλ} R^{μ ν}_{γ ε} R_{δξηλ} S_{βμν}}
```

Comparatively, this calculation exhibits a slow down of around 30%. Although such an execution penalty is regrettable in the instances when the final answer is unaffected by index class inclusion, such a slowdown is still well within the realm of acceptability.

When the tensor being canonicalized actually includes indices that are of differing index classes, then there can be a further speed penalty. This can be due to either working with an inherently

larger number of configurations, or possibly longer computation times for transpositional canonicalization, or possibly both conditions.

```
In[81]:= OptimizedCanonicalize[R^{aβγb}_{abγ} R^{ξλc}_{dβ} R^{ϵαd}_{cλ} S^{}_{ϵξ}] // Timing
Out[81]= {1.29333 Second, R^{αβγaδbcϵ}_{γδ} R^{ξd}_{abcϵ} S_{βξd}}
```

So our overall algorithm was some 20% slower in this last case, where class extended indices were necessary.

6.14 Linear Symmetries and the Complete Algorithm

Up until this stage we have avoided any discussion of linear symmetries in our canonicalization algorithms. Linear symmetries present a whole new dimension of complexity to the overall algorithm. Fortunately, the optimized canonicalization algorithm is fast enough that performing calculations with linear symmetries is eminently possible.

6.14.1 Origins of Linear Symmetries

The symmetries of tensors that have so far been mentioned are (*i*) transpositional symmetries, (*ii*) complex symmetries, and (*iii*) degeneracy symmetries. In practice, we must also take into account "linear" symmetries. We have purposely delayed the introduction of linear symmetries until this late stage since they are, in a certain sense, fundamentally different from the earlier symmetries. To motivate linear symmetries, consider the following equation which the Riemann tensor must obey.

$$\mathbf{R}_{\alpha\beta\gamma\delta} + \mathbf{R}_{\alpha\gamma\delta\beta} + \mathbf{R}_{\alpha\delta\beta\gamma} = 0 \tag{6.14.a}$$

Let us proceed to verify that this "linear" symmetry is indeed true. In principle, all that is needed is our canonicalization algorithm, together with the ability to resolve the Riemann tensor into terms involving Christoffel symbols and metric tensors.

We first add the notation for tensorial assignment, previously introduced in §3.5.3 *Dummy Indices*.

```
In[1]:= Notation[lhs_ :≔ rhs_ ⟺ TensorSetDelayed[lhs_, rhs_]];

In[2]:= (lhs_ :≔ rhs_) :=
          With[{dummies = DummyIndices[rhs] \ DummyIndices[lhs]},
            lhs := Module[dummies, rhs]]
```

As one will find in any text in general relativity [81, 240, 255, 326, 333], the Riemann tensor is expressible in terms of the metric and Christoffel symbols as follows.

```
In[3]:= (R_γ_ σ_ μ_ ν_)_hp :≔ g_γ λ (Γ^λ_σμ,ν − Γ^λ_σν,μ + Γ^α_σν Γ^λ_αμ − Γ^α_σμ Γ^λ_αν)
```

So as not to confuse the issues here, we have used a definition for the expansion of the Riemann tensor that is always active. After the involved work of §4 *Language Modifications*, we would expect that it would be possible to define this last rule just in a specific environment, say `ResolveR`. Indeed this is possible, and we will do almost exactly this in §7 *Tensor Calculus, Applications, and Quasi-Spin*. However, for now we are only interested in motivating linear symmetries, and so we forgo such elegance until later.

Returning to our verification of (6.14.a), observe that the linear sum of the cyclically rotated Riemann tensor terms is expanded just by entering them, due to the automatic expansion rule entered above.

```
In[4]:= R_αβγδ + R_αγδβ + R_αδβγ

Out[4]= g_αλ$1511 (Γ^α$1511_βδ Γ^λ$1511_α$1511γ −
                   Γ^α$1511_βγ Γ^λ$1511_α$1511δ + Γ^λ$1511_βγ,δ − Γ^λ$1511_βδ,γ) +
        g_αλ$1512 (−Γ^α$1512_γδ Γ^λ$1512_α$1512β + Γ^α$1512_γβ Γ^λ$1512_α$1512δ −
                   Γ^λ$1512_γβ,δ + Γ^λ$1512_γδ,β) +
        g_αλ$1513 (Γ^α$1513_δγ Γ^λ$1513_α$1513β − Γ^α$1513_δβ Γ^λ$1513_α$1513γ +
                   Γ^λ$1513_δβ,γ − Γ^λ$1513_δγ,β)
```

To transform this expression into a more recognizable form, we just expand the previous result and reindex it.

```
In[5]:= linearSumExpr = ReIndex @ Expand @ %

Out[5]= −g_αζ Γ^ϵ_γδ Γ^ζ_ϵβ + g_αζ Γ^ϵ_δγ Γ^ζ_ϵβ + g_αζ Γ^ϵ_βδ Γ^ζ_ϵγ − g_αζ Γ^ϵ_δβ Γ^ζ_ϵγ −
        g_αζ Γ^ϵ_βγ Γ^ζ_ϵδ + g_αζ Γ^ϵ_γβ Γ^ζ_ϵδ + g_αϵ Γ^ϵ_βγ,δ − g_αϵ Γ^ϵ_βδ,γ −
        g_αϵ Γ^ϵ_γβ,δ + g_αϵ Γ^ϵ_γδ,β + g_αϵ Γ^ϵ_δβ,γ − g_αϵ Γ^ϵ_δγ,β
```

Before using our canonicalization algorithm, we need to declare the symmetries of the various tensorial objects in the above expression (if we have not already done so). That is, we must declare that the Christoffel symbol is symmetric on its second and third indices, as well as declaring that the metric is symmetric.

```
In[6]:= DeclareSymmetries[Γ, 3, {(2 ↔ 3)_+}]
        DeclareSymmetries[g, 2, {(1 ↔ 2)_+}]
```

To show that the linear sum of permuted terms (6.14.a) is zero, we simply canonicalize the expanded sum.

```
In[8]:= OptimizedCanonicalize @ linearSumExpr

Out[8]= 0
```

Actually, we only gave the above expansion so that the reader would get a better understanding of what was actually occurring. We did not have to reindex the expressions at all, since our canonicalization algorithm takes them to canonical configurations in any case. Thus alternatively, we could have verified (6.14.a) as follows.

```
In[9]:= OptimizedCanonicalize @ Expand[R[α, β, γ, δ] + R[α, γ, δ, β] + R[α, δ, β, γ]]

Out[9]= 0
```

This shows that the linear sum of cyclically permuted terms must be 0. Equation (6.14.a) is an example of a linear symmetry that the Riemann tensor must obey. Another linear symmetry that the Riemann tensor must obey is the Bianchi identities [81, 240, 255, 326, 333].

$$\mathbf{R}_{\alpha\beta\gamma\delta;\sigma} + \mathbf{R}_{\alpha\beta\delta\sigma;\gamma} + \mathbf{R}_{\alpha\beta\sigma\gamma;\delta} = 0 \tag{6.14.b}$$

This relation can also be verified / derived in a similar way to that used to show (6.14.a). However, we will delay proving (6.14.b) until §7.3.2 *Linear Symmetries Revisited*, since we have not yet developed any of the needed forthcoming tools for the "calculus" aspects of tensors.

Another prominent tensor that has a linear symmetry is the Maxwell electromagnetic field tensor, F — see [81, 240, 255, 326, 333]. The F tensor is expressible in terms of the covariant derivatives of the four-vector potential by the following.

$$\mathbf{F}_{\mu\nu} = \mathbf{A}_{\mu;\nu} - \mathbf{A}_{\nu;\mu} \tag{6.14.c}$$

The tensor F satisfies the following linear symmetry.

$$\mathbf{F}_{\mu\nu,\alpha} + \mathbf{F}_{\nu\alpha,\mu} + \mathbf{F}_{\alpha\mu,\nu} = 0 \tag{6.14.d}$$

This linear relation is simple enough to easily verify with the tools so far developed. In an analogous way to defining the Riemann tensor above, let us express the partial derivative of the F tensor in terms of four-vector potentials.

```
In[10]:= F[μ_, ν_, α_] := A[μ, ν; α] - A[ν, μ; α]
```

And, by the definition of the covariant derivative, the following relation is also true.

```
In[11]:= A[α_, μ_, ν_] := A[α, μ, ν] - A[β, ν] Γ[β, α, μ] - A[β] Γ[β, α, μ, ν]
```

Our linear symmetry, (6.14.d), can thus be verified as follows.

```
In[12]:= F[μ ν , α] + F[ν α , μ] + F[α μ , ν]
```

$$\begin{aligned}\text{Out[12]=}\ & \mathbf{A}_{\alpha,\mu\nu} - \mathbf{A}_{\alpha,\nu\mu} - \mathbf{A}_{\mu,\alpha\nu} + \mathbf{A}_{\mu,\nu\alpha} + \mathbf{A}_{\nu,\alpha\mu} - \mathbf{A}_{\nu,\mu\alpha} - \\ & \mathbf{A}_{\beta\$1542,\alpha}\,\Gamma^{\beta\$1542}_{\mu\nu} + \mathbf{A}_{\beta\$1543,\alpha}\,\Gamma^{\beta\$1543}_{\nu\mu} - \mathbf{A}_{\beta\$1545,\mu}\,\Gamma^{\beta\$1545}_{\nu\alpha} + \\ & \mathbf{A}_{\beta\$1546,\mu}\,\Gamma^{\beta\$1546}_{\alpha\nu} - \mathbf{A}_{\beta\$1548,\nu}\,\Gamma^{\beta\$1548}_{\alpha\mu} + \mathbf{A}_{\beta\$1549,\nu}\,\Gamma^{\beta\$1549}_{\mu\alpha} - \\ & \mathbf{A}_{\beta\$1542}\,\Gamma^{\beta\$1542}_{\mu\nu,\alpha} + \mathbf{A}_{\beta\$1543}\,\Gamma^{\beta\$1543}_{\nu\mu,\alpha} - \mathbf{A}_{\beta\$1545}\,\Gamma^{\beta\$1545}_{\nu\alpha,\mu} + \\ & \mathbf{A}_{\beta\$1546}\,\Gamma^{\beta\$1546}_{\alpha\nu,\mu} - \mathbf{A}_{\beta\$1548}\,\Gamma^{\beta\$1548}_{\alpha\mu,\nu} + \mathbf{A}_{\beta\$1549}\,\Gamma^{\beta\$1549}_{\mu\alpha,\nu}\end{aligned}$$

As before, we can reindex this last result to obtain a more conventional looking expression.

```
In[13]:= ReIndex @ %
```

$$\begin{aligned}\text{Out[13]=}\ & \mathbf{A}_{\alpha,\mu\nu} - \mathbf{A}_{\alpha,\nu\mu} - \mathbf{A}_{\mu,\alpha\nu} + \mathbf{A}_{\mu,\nu\alpha} + \mathbf{A}_{\nu,\alpha\mu} - \mathbf{A}_{\nu,\mu\alpha} - \\ & \mathbf{A}_{\beta,\nu}\,\Gamma^{\beta}_{\alpha\mu} + \mathbf{A}_{\beta,\mu}\,\Gamma^{\beta}_{\alpha\nu} + \mathbf{A}_{\beta,\nu}\,\Gamma^{\beta}_{\mu\alpha} - \mathbf{A}_{\beta,\alpha}\,\Gamma^{\beta}_{\mu\nu} - \\ & \mathbf{A}_{\beta,\mu}\,\Gamma^{\beta}_{\nu\alpha} + \mathbf{A}_{\beta,\alpha}\,\Gamma^{\beta}_{\nu\mu} - \mathbf{A}_{\beta}\,\Gamma^{\beta}_{\alpha\mu,\nu} + \mathbf{A}_{\beta}\,\Gamma^{\beta}_{\alpha\nu,\mu} + \\ & \mathbf{A}_{\beta}\,\Gamma^{\beta}_{\mu\alpha,\nu} - \mathbf{A}_{\beta}\,\Gamma^{\beta}_{\mu\nu,\alpha} - \mathbf{A}_{\beta}\,\Gamma^{\beta}_{\nu\alpha,\mu} + \mathbf{A}_{\beta}\,\Gamma^{\beta}_{\nu\mu,\alpha}\end{aligned}$$

Finally, to verify the linear symmetry (6.14.d), we simply canonicalize the previous result.

```
In[14]:= OptimizedCanonicalize @ %
```

```
Out[14]= 0
```

The examples presented in this subsection have hopefully given some indication that linear symmetries of tensors frequently arise in practice. Generally, linear symmetries naturally arise from the particular expansions of tensors in terms of more "primitive" tensors. For instance, the Riemann tensor can be expanded in terms of Christoffel symbols and metric tensors; and consequently, due to the symmetries of these more "primitive" combined tensors, the linear symmetries of R arise. In a similar manner, because the electromagnetic tensor F is expressible in terms of the anti-symmetric sum of the partial derivatives of the four-vector potential, we can observe that there exists a linear symmetry of the tensor F.

Before we close this section, let us remove the direct expansion rule for the Riemann tensor previously introduced in this subsection.

```
In[15]:= (R[γ_ σ_ μ_ ν_])[hp] = .
```

Obviously, we somehow need to incorporate linear symmetries into our overall algorithm in some way. The next subsection presents the overall strategy we will use, and then the following subsections develop and discuss the material necessary to achieve the overall strategy.

6.14.2 Overview of the Linear Symmetries Algorithm

As we saw in the previous subsection, linear symmetries in tensors arise naturally in many circumstances. The question arises, however, of whether we actually need to change anything in our algorithms at all. For in the above discussions, all that was necessary was to express the tensors with linear symmetries in terms of more primitive tensors, and then we obtained results with these. Is this a viable strategy in practice? The answer is, unfortunately, no. Firstly, it is subject to combinatorial explosion; secondly, it forces all tensors to be expressed as primitive tensors; and thirdly, it will only yield the simplest forms if our tensor products sum to zero. Consequentially, to utilize linear symmetries, we need to modify our canonicalizing algorithms.

Our complete algorithm, `Canonicalize`, will loosely progress along the following lines. Given a tensor product to canonicalize, we first use a variant of our optimized canonicalization algorithm to canonicalize it with respect to the degenerate, complex, and transpositional symmetries. Then we generate, via a staged closure routine, all tensor products reachable via the linear symmetries and the corresponding *governing equations*. (We formally define and give examples of such things shortly.) We apply our optimized canonicalization algorithm to the configurations found at each stage, thus considerably reducing the number of configurations reachable via the linear symmetries. To avoid redundant recomputations and hence to further reduce computation times, we cache the relevant computations as we progress. Once we have generated all possible governing equations, we use a canonicalizing strategy to find the simplest form of our initial expression.

Before we step through an example of our complete algorithm, it is necessary to clarify what we mean by governing equations.

Definition 6.14.A: The set of *governing equations* for a tensor product or a tensorial expression is the maximal set of linear equations relating all possible canonical rearrangements reachable by the linear symmetries of the tensorial expression.

To relate the definition of a set of governing equations to a practical example, consider the tensor product $\mathbf{R}^{\alpha\beta\gamma\delta}\mathbf{R}_{\mu\nu\epsilon\tau}$. Acting with symmetry (6.14.a) on the first tensor factor in this tensor product yields the following governing equation.

$$\mathbf{R}^{\alpha\beta\gamma\delta}\mathbf{R}_{\mu\nu\epsilon\tau}+\mathbf{R}^{\alpha\gamma\delta\beta}\mathbf{R}_{\mu\nu\epsilon\tau}+\mathbf{R}^{\alpha\delta\beta\gamma}\mathbf{R}_{\mu\nu\epsilon\tau}=0$$

Actually, it is convenient to talk of *applying* a linear symmetry to a tensor to yield an equation. Thus, applying symmetry (6.14.a) to the second tensor factor, we obtain another governing equation.

$$\mathbf{R}^{\alpha\beta\gamma\delta}\mathbf{R}_{\mu\nu\epsilon\tau}+\mathbf{R}^{\alpha\beta\gamma\delta}\mathbf{R}_{\mu\epsilon\tau\nu}+\mathbf{R}^{\alpha\beta\gamma\delta}\mathbf{R}_{\mu\tau\nu\epsilon}=0$$

Each of these equations has new tensor products in it, from which we could generate further governing equations. For instance, we could use the symmetry (6.14.a) on the first tensor factor of $\mathbf{R}^{\alpha\beta\gamma\delta}\,\mathbf{R}_{\mu\epsilon\tau\nu}$ to yield yet another independent governing equation. If we continue this process, we will eventually end up with a finite set of equations which cannot be added to by symmetry (6.14.a). This set would be the set of governing equations for the given tensor product under the given linear symmetry (6.14.a).

In formal considerations, it might be important to clarify the distinction between an underlying linear symmetry and its equation form. This situation is almost identical to that stated in §6.2.1 *Terminology and Background*. Namely, our signed permutations or transpositions are equivalent to equations. For instance, $\mathbf{R}_{ijkl} == -\mathbf{R}_{jikl}$ is equivalent to $(1 \leftrightarrow 2)_-$, and $\mathbf{R}_{ijkl} == \mathbf{R}_{klij}$ is equivalent to $\{3, 4, 1, 2\}_+$. In almost this exact same way, there is a correspondence between an *"equational linear symmetry"* and an underlying *"formal linear symmetry"*. The theoretical considerations concerning linear symmetries, to be presented in the next five subsections, are probably best described in terms of equational linear symmetries rather than formal linear symmetries. Thus, we need not concern ourselves with the formal underlying linear symmetry representations until §6.14.8 *Linear Symmetry Permutations*. In any case, an equational linear symmetry is isomorphic to its underlying formal linear symmetry, so we will largely ignore the distinction.

Now that we understand the concept of the set of governing equations, let us return to our complete algorithm. Although our brief description above may sound somewhat complex, to carry it out is actually a relatively simple procedure. Let us trace through a small motivating example. Say we are trying to canonicalize the following expression.

```
In[16]:= expr = R[α, λ, τ, σ] R[δ, σ, ϵ, γ] - R[α, β, λ, τ] R[δ, ϵ, β, γ] + R[α, λ, β, τ] R[δ, β, γ, ϵ]
```

$$\mathbf{R}^{\alpha\ \lambda\ \tau\,\sigma}_{\delta\ \sigma\ \epsilon\,\gamma}\,\mathbf{R}^{\ \ \ \

```
Out[18]= {R[α β λ, δ, τ, β γ ε] - R[α λ β, δ, τ, β γ ε] == R[α, λ β, δ, τ, β γ ε],
  R[α β λ, δ, τ, γ β ε] - R[α, λ β, δ, τ, γ β ε] == R[α λ β, δ, τ, γ β ε],
  R[α, λ β, δ, τ, γ β ε] - R[α, λ β, δ, τ, ε β γ] == R[α, λ β, δ, τ, β γ ε],
  R[α β λ, δ, τ, γ β ε] - R[α β λ, δ, τ, ε β γ] == R[α β λ, δ, τ, β γ ε],
  R[α λ β, δ, τ, γ β ε] - R[α λ β, δ, τ, ε β γ] == R[α λ β, δ, τ, β γ ε],
  R[α β λ, δ, τ, ε β γ] - R[α λ β, δ, τ, ε β γ] == R[α, λ β, δ, τ, ε β γ]}
```

The full tensorial canonicalization algorithm must then "simplify" the given expression with respect to the governing equations. To demonstrate this, let us use the full canonicalization algorithm `Canonicalize`. This algorithm simplifies with respect to linear symmetries in addition to all of the standard symmetries.

```
In[19]:= Canonicalize[expr]
Out[19]= -R[α β λ, δ, τ, γ β ε] + R[α λ β, δ, τ, γ β ε]
```

The overall process above contains the essence of how our complete algorithm, `Canonicalize`, operates. We will discuss more of the details pertaining to the function `Canonicalize` as we progress through this section.

The next subsection discusses what is meant by a canonical form for a system of equations. Then in §6.14.4 *Gröbner Bases*, we briefly present the background necessary to use Gröbner bases to find canonical forms under our linear symmetry equations. In that section we rely on the known results involving Gröbner bases to show that we will always obtain a canonical form using such bases. Then in §6.14.5 *Equational Systems and Gröbner Canonicalization*, we re-examine the governing equations arising from our example tensorial expression, and step through the Gröbner canonicalization process. That subsection also illustrates how term orderings affect canonical forms. The next subsection investigates changing the term ordering and the attendant consequences. Following this, we introduce the method of direct reduction, which is an alternative to Gröbner canonicalization. The representation of our linear symmetry objects and auxiliary equations are then covered in §6.14.8 *Linear Symmetry Permutations*. We finally briefly address implementation issues and then conclude with some example timings of our canonical algorithm.

We can summarize this subsection as follows. The problem outstanding to us is the reduction of an expression to a "simplest form" with respect to a set of governing equations. The remainder of this section deals with this problem.

6.14.3 Linear Symmetries and Canonical Functions

In defining a canonicalization routine that includes linear symmetries, we have several choices to make in the determination of a complete specification of a canonical form. Also, as always, we must ensure our canonical form is unique. And of course, we would like our canonical form to be as "simple" as possible. Unfortunately, there are several competing notions of "simple" when considering expressions involving tensors with linear relations.

Probably the most obvious choice for a canonical form would be to select the expression amongst all equivalent expressions with the smallest number of terms. There are several ways we could handle the case when different expressions have the same number of terms. We could easily create a selection criterion which could be shown to yield a canonical form. However, as far as the author is aware, it is not possible to find this particular canonical form efficiently. It is highly likely that finding this canonical form would require an NP complete algorithm — see Garey & Johnson [122]. Our approach to finding a canonical form will be to use Gröbner bases, even though we are dealing with linear equations.

Technical Note: Actually, as previously mentioned, any deterministic "referentially transparent" function[125, 22, 264, 310] that acts on a canonical function yields a new canonical function. We could utilize this knowledge by initially using the methods forthcoming in this section to find a canonical form for a tensorial expression, and then proceed to deterministically optimize the canonical form with respect to the governing equations to obtain a new optimized canonical form. Thus, any two expressions that are equivalent would first be reduced to exactly the same initial canonical form; and then, by deterministic optimization, both would be taken to exactly the same optimized canonical form. That is, our post-canonical deterministic optimization would maintain "canonical-ness". Hence, all that would remain would be to find a deterministic optimizer. There are many different possibilities, but a likely candidate for this kind of NP-hard problem would be to use something along the lines of *simulated annealing* [269, 300]. However in this case, we would use *deterministic annealing* — see[97, 98]. Such refinements have not been made since the canonical forms generated by our forthcoming methods are quite acceptable (in the author's opinion.)

Before launching into more complicated discussions involving Gröbner bases, let us explain why a more generic *Mathematica* function such as `Solve` cannot be used. The pivotal reason is that `Solve` is *not* a canonicalizing function. Let us demonstrate this and then discuss the implications of this important realization.

To show that `Solve` is not a canonicalizing function, we will use `Solve` to find answers in two simple sets of equations. These equations will be set up so that the answers are mathematically equivalent, given the equations, but `Solve` will return different answers. In this way we can see that `Solve` will violate property (*ii*) of the definition of a canonicalizing function, as given in Definition 6.7.A. Specifically, consider solving for x in the following sets of equations involving a, b, c : $\{a+b+c==0,\ x==-a\}$ and $\{a+b+c==0,\ x==b+c\}$

```
In[20]:= Solve[{a + b + c == 0, x == -a}, x]

Out[20]= {{x → -a}}

In[21]:= Solve[{a + b + c == 0, x == b + c}, x]
```

```
Out[21]= {{x → b + c}}
```

By inspection, it is obvious that x is semantically equal to both $-a$ and to $b + c$, yet the results of `Solve` are not identical. Therefore, we can say that `Solve` is *not* a canonicalizing function. Indeed, `Reduce` is also not a canonicalizing function.

```
In[22]:= Reduce[{a + b + c == 0, x == -a}]

Out[22]= b == -a - c && x == -a

In[23]:= Reduce[{a + b + c == 0, x == b + c}]

Out[23]= a == -b - c && x == b + c
```

This leaves us in a rather awkward position. We cannot use *Mathematica*'s standard functions `Solve` or `Reduce` to determine a canonical form for expressions involving linear symmetries. Thus we are forced to turn to a different solution to this problem, namely, the use of Gröbner bases.

6.14.4 Gröbner Bases

Gröbner bases were discovered by Bruno Buchberger in his doctoral thesis[34]. (He named these bases after his supervisor, Gröbner.) Without doubt Gröbner bases, and the Buchberger algorithm to find such bases, were seminal discoveries. Every major computer algebra system today uses some variation on the Buchberger algorithm to generate Gröbner bases from a given set of polynomials. Indeed, the field of research has bifurcated into others; for instance, see Buchberger & Winkler [37]. The report by Heck[158] or the chapter in Geddes[124] provides a fast and readable introduction to the subject, along with some applications. In addition, the following books, amongst others, give detailed information on the theory of Gröbner bases: Froberg[117], Adams & Lousraunau[5], and Becker & Weispfenning[19].

Although in this thesis we will not present any of the theory of Gröbner bases, and indeed we will only use Gröbner bases in the capacity of a tool, let us briefly say a few words about them. The following discussion is an abbreviated and tailored form of the introductions of Geddes[124] and Heck[158].

Recall from basic ring theory (see Fraleigh[114], MacLane[216], Froberg[117], etc.) that an ideal I of a commutative ring R with identity obeys the following:

(*i*) $I \subseteq R$
(*ii*) $p, q \in I \Rightarrow p - q \in I$
(*iii*) $p \in I, r \in R \Rightarrow rp \in I$

Consider a finite set of polynomials $P = \{p_1, \ldots, p_k\}$ in $x_1, \ldots, x_n$ over some field F, that is, $P \subset F[x_1, \ldots, x_n]$. It is a textbook exercise to check that all possible linear combinations of these polynomials over the ring of polynomials form an ideal. That is, the infinite set $I = \{\sum a_i p_i \mid p_i \in P,\ a_i \in F[x_1, \ldots, x_n]\}$ is an ideal of the ring of polynomials $F[x_1, \ldots, x_n]$. We say that the set P *generates* I, that is, $I = \langle P \rangle$, or equivalently P is a *basis* for the polynomial ideal I.

Let us illustrate some of the relevant uses of polynomial ideals through examples. Following Geddes[124], consider the simple set of polynomials in $\mathbb{Q}[x, y, z]$

```
In[24]:= p1 = x^3 y z - x z^2;  p2 = x y^2 z - x y z;  p3 = x^2 y^2 - z^2;
```

These polynomials generate a polynomial ideal in $\mathbb{Q}[x, y, z]$, namely $\langle p_1, p_2, p_3\rangle = \{ap_1 + bp_2 + cp_3 \mid a, b, c \in \mathbb{Q}[x, y, z]\}$. Herein a natural question arises. How can one determine if a given polynomial, say q, is in the ideal $\langle p_1, p_2, p_3\rangle$? For example, is $q = x^2 yz - z^3$ in $\langle p_1, p_2, p_3\rangle$? It turns out that it is.

```
In[25]:= x^2 y z - z^3 == Expand[-x p2 + z p3]

Out[25]= True
```

In general, however, the membership problem had been difficult to solve until the advent of Gröbner bases. The question of membership is equivalent to deciding whether q simplifies to 0 under the equations $p_1 = 0$, $p_2 = 0$, $p_3 = 0$.

Related to the membership problem is the following more relevant question. Can we reduce an expression modulo a polynomial ideal? For instance, 17 mod 10 is simply 7. Similarly then, what is $3xz^7 + xy^6z^4 - y^2x$ mod $\langle p_1, p_2, p_3\rangle$? Or, equivalently, is there a simpler form for $3xz^7 + xy^6z^4 - y^2x$, given that $p_1 = 0$, $p_2 = 0$, $p_3 = 0$? One can verify that indeed there is a simpler form: $-xy^2 + 4xz^2$.

```
In[26]:= Simplify[z^4 x y^6 + 3 x z^7 - y^2 x == -x y^2 + 4 x z^2, p1 == 0 ∧ p2 == 0 ∧ p3 == 0]

Out[26]= True
```

Clearly, in some intuitive way, which we will soon formalize, $-xy^2 + 4xz^2$ is "smaller" than $z^4xy^6 + 3xz^7 - y^2x$. We will shortly discuss such a reduction of a polynomial with respect to an ideal (equivalently, a system of equations) using Gröbner bases.

In essence a Gröbner basis for a polynomial ideal I is an alternative set of generators for I that has "nice" properties. Given a set of generators P for I, the Buchberger algorithm calculates a new set of "nice" generators G such that $\langle G\rangle = \langle P\rangle = I$. An important property of a Gröbner basis $G = \{g_1, \ldots, g_n\}$ for an ideal $I = \langle P\rangle$ is that it shares the same set of zeros as the original polynomials in P. G is "nice" in the sense that the system of equations $g_1 = 0, \ldots, g_n = 0$ is easier to solve than the original system $p_1 = 0, \ldots, p_m = 0$ (assuming the latter is solvable).

For instance, the Gröbner basis of $\langle p_1, p_2, p_3\rangle$ is the following:

```
In[27]:= G = GroebnerBasis[{p1, p2, p3}]

Out[27]= {z^4 - z^5, -z^3 + y z^3, x z^2 - x z^3, -x z^2 + x y z^2,
            -x y z + x y^2 z, -x^2 z^2 + z^4, -x^2 y z + z^3, x^2 y^2 - z^2}
```

Although the members of G might not look "nice" on first sight, we can see that the first polynomial $z^4 - z^5$ is entirely in z alone, and so we can solve $z^4 - z^5 = 0$ for z.

```
In[28]:= Solve[z^4 - z^5 == 0, z]

Out[28]= {{z → 0}, {z → 0}, {z → 0}, {z → 0}, {z → 1}}
```

These solutions for z can be substituted back into the system of equations allowing us to proceed and solve $-z^3 + y z^3 = 0$ for y, etc. In this way we can solve the original non-linear system of polynomials $p_1 = 0$, $p_2 = 0$, $p_3 = 0$ for its zeros.

The Buchberger algorithm[34] which generates a Gröbner basis is based on reducing a polynomial with respect to other polynomials to yield "smaller" polynomials. To characterize "smaller", an ordering on polynomials needs to be introduced, much as we introduced an ordering on configurations in §6.6 *The Ordering of Configurations*. This ordering, in the nomenclature of the field, is a *term ordering*. A term is just a product of powers of indeterminants, that is, something of the form $x^i y^j \ldots z^k$ for some indeterminants $x, y, \ldots, z$ and $i, j, \ldots, k \in \mathbb{N}$. Gröbner bases are calculated relative to a term ordering. There are many different possible term orderings. For instance, the following is standard in the literature.

Definition 6.14.B: Assuming $x_1 \succ x_2 \succ \ldots$, the *pure lexicographic term ordering* on monomials is determined as follows. ${x_1}^{i_1} {x_2}^{i_2} \ldots {x_n}^{i_n} \prec {x_1}^{j_1} {x_2}^{j_2} \ldots {x_n}^{j_n}$ if and only if $\exists\, l \in \{1, \ldots, n-1\}$, such that $\forall\, k < l$ it is the case that $i_k = j_k$ and $i_l < j_l$.

So, for example, the pure lexicographic term ordering with $x \succ y \succ z$ results in the following.

$$z \prec z^2 \prec \ldots \prec y \prec yz \prec yz^2 \prec \ldots \prec y^2 \prec \ldots \prec x \prec \ldots$$

Once we have an ordering on terms, we can reduce one polynomial with respect to a set of others. The Gröbner basis G above was calculated with respect to the default lexicographic ordering. Let us reduce a polynomial term with respect to it.

```
In[29]:= PolynomialReduce[x^2 y^8 z^9, G, {x, y, z}]

Out[29]= {{-x^2 y^8 - x^2 y^8 z - x^2 y^8 z^2 - x^2 y^8 z^3 - x^2 y^8 z^4,
            x^2 z + x^2 y z + x^2 y^2 z + x^2 y^3 z + x^2 y^4 z + x^2 y^5 z + x^2 y^6 z + x^2 y^7 z,
            -x - x z, 0, 0, -1, 0, 0}, z^4}
```

`PolynomialReduce` expresses a polynomial q as $a_1\, g_1 + a_2\, g_2 + \ldots + a_i\, g_i + rem$ and returns $\{\{a_1, a_2, \ldots, a_n\}, rem\}$, where the a_i are polynomials and *rem* is minimal with respect to the term ordering. By minimal we mean there is no other combination of a_i's which would yield a "smaller" remainder *rem*. We say that q reduces to *rem* under G, or $q \longrightarrow_G rem$, or equivalently *rem* = reduce(q, G). For instance, the last result showed that $x^2 y^8 z^9 \longrightarrow_G z^4$, that is, $x^2 y^8 z^9$ equals z^4 under G. This affirms our intentions in that under the standard lexicographic ordering, $x \succ y \succ z$, it is clear that $x^2 y^8 z^9 \succ z^4$, thus indeed we have "reduced" the expression. In fact, if we changed the ordering of terms, we could reduce the expression to a different form. For example, if we use the reverse lexicographic term ordering, $z \succ y \succ x$, we obtain:

```
In[30]:= G_reverse = GroebnerBasis[{p1, p2, p3}, {z, y, x}]

Out[30]= {x^3 y^2 - x^5 y^2, x^3 y^2 - x^3 y^3, x^3 y^2 - x^3 y z, -x y z + x y^2 z, x^2 y^2 - z^2}

In[31]:= PolynomialReduce[x^2 y^8 z^9, G_reverse, {z, y, x}]

Out[31]= {{0, -x - x z - x z^2 - x z^3 - x z^4 - x z^5 - x z^6 - x z^7,
            -x y - x y z - x y z^2 - x y z^3 - x y z^4 - x y z^5 - x y z^6, x z^8 + x y z^8 +
              x y^2 z^8 + x y^3 z^8 + x y^4 z^8 + x y^5 z^8 + x y^6 z^8, -x^2 y z^7}, x^4 y^2}
```

This illustrates several important points. Firstly, Gröbner bases with different term orderings can be substantially different; indeed G has 7 basis polynomials whereas $G_{reverse}$ has only 5. Computation times can also very dramatically for the different term orderings. Secondly, reducing polynomials with respect to different term orderings leads to different answers, even though the ideals must be the same. For instance even though $\langle G \rangle = \langle G_{reverse} \rangle$, the reduction $x^2\, y^8\, z^9 \longrightarrow_G z^4$ is different to the reduction $x^2\, y^8\, z^9 \longrightarrow_{G_{reverse}} x^4\, y^2$. Incidentally, one might feel that $x^4\, y^2$ is not "smaller" than $x^2\, y^8\, z^9$ since the power of x is greater in the term $x^4\, y^2$; but this is solely due to the fact that in the reverse lexicographic term ordering $x \prec y \prec z$, we have $x^4\, y^2 \prec x^2\, y^8\, z^9$ since the power of z is smaller in $x^4\, y^2$ than in $x^2\, y^8\, z^9$.

It is fairly simple to specify an algorithm for polynomial reduction, but we do not do so here (see any of the above references for details). For completeness, some texts use the terminology *normalForm*(q, G) in place of *reduce*(q, G). For reference, let us give the definition of a Gröbner basis.

Definition 6.14.C: An ideal basis $G \subset F[x_1, \ldots, x_n]$ is a *Gröbner basis* with respect to an admissible term ordering if and only if $p \longrightarrow_G 0$ for all $p \in \langle G \rangle$.

We will not delve into any details concerning the many interesting and useful theorems about Gröbner bases. However, there are two theorems in particular that we need to consider.

Theorem 6.14.A: If G is a Gröbner basis, then:

$$reduce(p, G) = reduce(q, G) \text{ if and only if } p \underset{\langle G \rangle}{\overset{mod}{\equiv}} q, \text{ that is, } p - q \in \langle G \rangle$$

Proof: See Buchberger [34], or any of the other references on Gröbner bases, such as Geddes[124] etc. ■

Theorem 6.14.A is the main result we need. Reduction modulo a Gröbner basis yields a canonical form since reduction clearly satisfies the requirements of Definition 6.7.A. That is, (*i*) $reduce(q, G) \underset{\langle G \rangle}{\overset{mod}{\equiv}} q$ and (*ii*) $p \underset{\langle G \rangle}{\overset{mod}{\equiv}} q \Rightarrow reduce(p, G) \equiv reduce(q, G)$. Thus the reduced form is a canonical form.

In our canonicalization algorithm, if we have linear symmetries, then we will generate systems of equations from a given tensor expression consisting of sums of tensor products. We can then form a Gröbner basis from the system of equations, and thus reduce the tensor expression with respect to this basis to yield a canonical form for the tensor expression. Moreover, we do not have to worry about how or in what order we assemble the system of equations due to the following definition and theorem.

Definition 6.14.D: A set $G \subset F[x_1, \ldots, x_n]$ is *reduced* if $g = reduce(g, G \setminus \{g\})$ for all $g \in G$. The set G is *monic* if for every $g \in G$ the coefficient of the leading term (largest term) in g is 1.

Theorem 6.14.B: If G, H are reduced monic Gröbner bases such that $\langle G \rangle = \langle H \rangle$, then $G = H$.

Proof: See Buchberger [35]. ■

Owing to this theorem, it does not matter in what way we assemble the defining symmetry equations, since the Gröbner basis is guaranteed to be unique. Of course, reductions still depend upon the term ordering. Term orderings are the topic we will be especially interested in since they will dictate the final mold of our canonical forms.

Technical Note: Actually, given a Gröbner basis with respect to one term ordering, it is possible to efficiently find a Gröbner basis to the same polynomial ideal but with respect to a different term ordering. A notable method to do this is the recently discovered "Gröbner walk" — see Collart [71] or Amrhein[8]. Indeed, to find a lexicographic Gröbner basis, it can be considerably more efficient to first find a total degree order Gröbner basis and then convert this to a lexicographic Gröbner basis via a "Gröbner walk".

Before we close this section it should be mentioned that since we will only be using linear polynomials, we have no fundamental need for the full theory of Gröbner bases and instead could content ourselves with merely using Gaussian elimination. However, the Gröbner basis generation algorithm implemented in *Mathematica* defaults to Gaussian elimination in any case, and the reduction to a canonical form is the pivotal property that is proved in many texts for Gröbner bases. Moreover, the formalism of Gröbner bases is more fitting to our description of the problem.

6.14.5 Equational Systems and Gröbner Canonicalization

The previous subsection presented a brief overview of Gröbner bases. In it we saw that reducing an expression with respect to a Gröbner basis was a canonicalizing operation. However, the term ordering we chose was important in determining the final answer. In this section we will give concrete examples using the Gröbner basis method to find canonical forms with respect to a set of equations and a term ordering.

Let us first illustrate why term orderings are important with an example. Recall the tensorial expression we used in our motivating examples in §6.14.2 *Overview of the Linear Symmetries Algorithm*.

```
In[32]:= expr = R[α, λ, τ, σ]_{δ, σ, ε, γ} - R[α, β, λ, τ]_{δ, ε, β, γ} + R[α, λ, β, τ]_{δ, β, γ, ε};
```

We canonicalized *expr* to yield a sum of two tensor products.

```
In[33]:= Canonicalize[expr]

Out[33]= -R[α, β, λ, τ]_{δ, γ, β, ε} + R[α, λ, β, τ]_{δ, γ, β, ε}
```

However, somewhat perturbingly, our original expression *expr* is in fact expressible as a single tensor product.

```
In[34]:= Canonicalize[expr == -R[α, λ, β] R[τ] ]
                              δ      γ β ε

Out[34]= True
```

Why then does our `Canonicalize` function return an answer that is obviously not as simple as a single tensor product? The answer is that, unfortunately, we do not have an efficient general algorithm which finds the shortest form for our expressions. This was previously mentioned in §6.14.3 *Linear Symmetries and Canonical Functions*. We will elaborate on this shortly. First though, let us investigate what underlies the answer that `Canonicalize` returns.

Internally the canonicalization algorithm generates all its equations in terms of configurations. The encoding and reconstitution phases of our algorithm occur only at the beginning and the end of the overall process. In fact, each configuration which is canonical with respect to the standard symmetries — that is, $\mathcal{S}_X$, $\mathcal{S}_D$, and $\mathcal{S}_T$ — is stored as a symbol. This greatly simplifies the manipulation of the governing equations. To view our set of equations, let us use the function `tensorProductToTensorSymbol` (defined in the *Tensors* package), which transforms from a canonical tensor product to a tensor symbol.

```
In[35]:= toTensorSymbols @ tensors_Plus := toTensorSymbols /@ tensors;
         toTensorSymbols [ nt_ tensorProduct_ ] :=
            nt toTensorSymbols @ tensorProduct /; FreeQ[nt, Tensor];
         toTensorSymbols @ eqn_Equal := toTensorSymbols /@ eqn;
         toTensorSymbols @ other_ := tensorProductToTensorSymbol @ other;
```

We can now view and manipulate the governing equations for our tensorial expression *expr*.

```
In[39]:= equations = toTensorSymbols /@ EquationsOfExpression @ expr

Out[39]= {TPS`Π2 - TPS`Π3 == TPS`Π1, TPS`Π5 - TPS`Π6 == TPS`Π4,
            TPS`Π6 - TPS`Π7 == TPS`Π1, TPS`Π5 - TPS`Π8 == TPS`Π2,
            TPS`Π4 - TPS`Π9 == TPS`Π3, TPS`Π8 - TPS`Π9 == TPS`Π7}
```

The symbols in the equations are defined in the context `TPS`. This separate context is intended to be suggestive of "Tensor Product Symbol", and has been used in order to avoid polluting the global name space.

Our original expression can be reduced to canonical symbols.

```
In[40]:= canonicalSymbols = toTensorSymbols /@ OptimizedCanonicalize @ expr

Out[40]= -TPS`Π1 - TPS`Π8 + TPS`Π9
```

We now have an expression, that is `-TPS`Π1-TPS`Π8+TPS`Π9`, which we would like to reduce with respect to a set of equations. This exact topic was covered in the previous subsection. Thus, let us proceed by applying the methods presented therein to reduce this expression.

The Gröbner basis method depends on a term ordering, which in this case means an ordering on the symbols in our expressions. Yet these symbols represent signed configurations, on

which we have an almost total ordering. Moreover, the tensor package actually caches just the positive signed configurations, thus all of the symbols represent positive signed configurations. (That is why minus signs have arisen in some of our equations, for instance `TPS`Π2-TPS`Π3 == TPS`Π1`.) Since there is a total ordering on the positive signed configurations, we can use this ordering to naturally generate a *derived* term ordering on the tensor symbols.

```
In[41]:= orderedTensorSymbols = Reverse @ sortTensorSymbols @
                  Union @ Cases[{equations}, symb_?tensorProductSymbolQ, {0, ∞}]

Out[41]= {TPS`Π7, TPS`Π8, TPS`Π6, TPS`Π1,
          TPS`Π5, TPS`Π2, TPS`Π9, TPS`Π4, TPS`Π3}
```

Technical Note: Both `sortTensorSymbols` and `tensorProductSymbolQ` are defined in the full source code for the canonicalization algorithm. Their meanings should be self evident.

We can now directly apply the Gröbner basis reduction methods of the previous subsection.

```
In[42]:= Last @ PolynomialReduce[canonicalSymbols,
             GroebnerBasis[equations, orderedTensorSymbols], orderedTensorSymbols]

Out[42]= TPS`Π4 - TPS`Π5
```

If we transform each of these symbols back to a tensor product, we obtain exactly the same result as was obtained by applying `Canonicalize` to *expr*.

```
In[43]:= % /. term_?tensorProductSymbolQ :> tensorSymbolToTensorProduct @ term
```

$$\text{Out[43]=}\quad -\mathbf{R}^{\alpha\beta\lambda}{}_{\delta}\,\mathbf{R}^{\tau}{}_{\gamma\beta\epsilon}+\mathbf{R}^{\alpha\lambda\beta}{}_{\delta}\,\mathbf{R}^{\tau}{}_{\gamma\beta\epsilon}$$

```
In[44]:= Canonicalize[expr] == %

Out[44]= True
```

Thus we have stepped through almost exactly the same process that the full algorithm follows. That is, we have stepped through the algorithm embodied in `Canonicalize`. Let us now progress onto examining how this algorithm depends on term orderings in practice.

6.14.6 Term Orderings

Now that we have a deeper understanding of how our algorithm functions, let us return to the issue of term ordering. Because our algorithm reduces expressions with respect to a Gröbner Basis, we do not always obtain the "simplest" equivalent form for a given tensorial expression. In fact this is exactly what we observed at the beginning of the previous subsection. That is, we found a single tensor product which upon canonicalization became the sum of two tensor products.

```
In[45]:= Canonicalize[-R^α λβ τ
                        δ      γβε]
```

$$\text{Out[45]=}\quad -\mathbf{R}^{\alpha\beta\lambda}{}_{\delta}\,\mathbf{R}^{\tau}{}_{\gamma\beta\epsilon}+\mathbf{R}^{\alpha\lambda\beta}{}_{\delta}\,\mathbf{R}^{\tau}{}_{\gamma\beta\epsilon}$$

Admittedly, each of the terms is more "canonical" than the original, but this is of little consolation since now there are two terms instead of one. Naturally this begs the following question. In general, does it often occur that canonicalizing a single term will yield an answer which consists of many terms? Fortunately the answer is "mostly no". Usually the result is limited to just a few terms. We will revisit this in appendix §C.3.3 *Multiple Reductions*. It is illuminating though to observe this phenomenon in the context of symbols and equations rather than for full blown tensor products.

```
In[46]:= toTensorSymbols[-R^α  λβ  R^τ
                          δ      γβε]
```

```
Out[46]= -TPS`Π6
```

Reducing `TPS`Π6` with respect to our Gröbner basis and the derived term ordering we obtain the following.

```
In[47]:= Last @ PolynomialReduce[TPS`Π6,
              GroebnerBasis[equations, orderedTensorSymbols], orderedTensorSymbols]
```

```
Out[47]= -TPS`Π4 + TPS`Π5
```

This of course exactly parallels the results returned using the full tensor products. However, since we have used product symbols we can simply change the term ordering and repeat the calculation. If we make `TPS`Π6` the least important term in our ordering, then reducing `TPS`Π6` should effect no change.

```
In[48]:= orderedTensorSymbols2 = {TPS`Π7, TPS`Π8, TPS`Π1,
                  TPS`Π5, TPS`Π2, TPS`Π9, TPS`Π4, TPS`Π3, TPS`Π6};
```

```
In[49]:= Last @ PolynomialReduce[TPS`Π6,
              GroebnerBasis[equations, orderedTensorSymbols2], orderedTensorSymbols2]
```

```
Out[49]= TPS`Π6
```

It must also be the case then that anything equivalent to `TPS`Π6` will be reduced to `TPS`Π6`. Indeed we can confirm this for the equivalent expression `-TPS`Π4+TPS`Π5`.

```
In[50]:= Last @ PolynomialReduce[-TPS`Π4 + TPS`Π5,
              GroebnerBasis[equations, orderedTensorSymbols2], orderedTensorSymbols2]
```

```
Out[50]= TPS`Π6
```

Thus we see that term orderings are critically important in determining the "simplicity" of our final result. Obviously this raises the question of whether there exists an optimal term ordering? The answer is in general no: the optimal term ordering depends on both the governing equations and the expression to be canonicalized.

It is important to point out that the various Gröbner bases with respect to the different term orderings are in fact distinct.

```
In[51]:= Length /@ GroebnerBasis[equations, orderedTensorSymbols]
```

```
Out[51]= {3, 3, 3, 3, 5}
```

```
In[52]:= Length /@ GroebnerBasis[equations, orderedTensorSymbols2 ]

Out[52]= {3, 3, 3, 4, 4}
```

Given all of the above, the next question we could ask is the following. Why not try all of the different orderings? Although this is theoretically possible, in practice it becomes infeasible due to combinatorial explosion. For instance, for the simple system we have just considered there are 9!=362880 different term orderings. We could possibly reduce this by several orders of magnitude by some careful considerations, but the point would be moot. The generation of a Gröbner basis for each term ordering takes a significant amount of time. So much so, that for some of our more complex problems it would be unacceptable to try even 100 different orderings. (To corroborate this we will later give some timings of finding the Gröbner basis for larger sets of equations. This is despite the fact that given a Gröbner basis for a set of equations with a given term ordering, there are special methods whereby one can find the Gröbner basis for the equations with respect to a new term ordering — see Collart [71], Amrhein[8], Froberg[117], Adams & Lousraunau[5], and Becker & Weispfenning[19].)

Technical Note: As a related item, finding the "best" term ordering for a Gröbner basis is an NP-complete problem [122].

Motivated by the ideas in this subsection and the fact that we appear to have some latitude in choosing the term ordering, we can then ponder whether alternatives exist to the Gröbner basis method in our special case. This is the topic of the next subsection.

6.14.7 The Method of Direct Reduction

Given the intricacies of the methods described previously, one might think that there may be other, simpler ways of getting to the canonical form for a sum of tensor polynomial terms. Indeed, we cannot rule out the possibility that a simpler way exists. Actually, it would be ideal if a simpler provable method could be found that covered all of the relevant cases. Such a canonicalizing algorithm might be made possible since we are dealing only with linear equations.

For the linear symmetries introduced so far, each of the final equations come out in the form $a+b+c=0$. The first and most obvious algorithm is thus the following. If we already have a term ordering on the various polynomial parts of our canonicalized tensor, we could just reduce the highest order term to the relevant combination of the lower order terms. That is, in this case if $a \prec b \prec c$, then we could transform $c \to -a-b$. Let us call this method direct reduction. This can be formally stated in the following definitions.

Definition 6.14.E: Given a linear equation $c_1\, t_1 + c_2\, t_2 + \ldots + c_n\, t_n = 0$ and a term ordering $\prec$ on the terms in the equation, then the *induced reduction rule* is the rule $t_l \to -(\sum_{i \neq l} t_i)/c_l$, where t_l is the leading term in the equation.

Definition 6.14.F: Assume H is a set of linear governing equations on the terms $t_1, \ldots, t_n$, and also that $\prec$ is a term ordering on the terms. Relative to $\prec$, the *reduction rules* induced by H consist of all the reduction rules induced by each equation in H.

The method of direct reduction just consists of reducing each term in an expression with respect to a set of reduction rules induced by a set of governing equations. Formally this is stated as follows.

Definition 6.14.G: Assume H is a set of linear governing equations on the terms $t_1, \ldots, t_n$, and also that $\prec$ is a term ordering on the terms. Given an expression *expr*, then *the reduced form* of *expr* is given by repeatedly applying all replacement rules in the set of *reduction rules* induced by H. We will equivalently refer to this as the *method of direct reduction*.

Unfortunately, in general the direct reduction strategy is not always a canonicalizing one. That is, two equivalent expressions are not always reduced to the same canonical form — see Definition 6.7.A. A simple counterexample, showing that direct reduction is not a canonicalizing function, is provided by the following system. It violates (*ii*) of Definition 6.7.A.

$$\begin{aligned} &a+b+f==0 \\ &c+d+f==0 \\ &a \prec b \prec c \prec d \prec f \end{aligned} \tag{6.14.e}$$

Now consider the expression $a+b-c-d$. This is clearly equivalent to $-f+f$ or 0. However, our reduction strategy will not alter any of the terms in the equations. Thus at first, it appears that the simple strategy of just reducing any higher order terms to combinations of lower order terms is flawed beyond redemption.

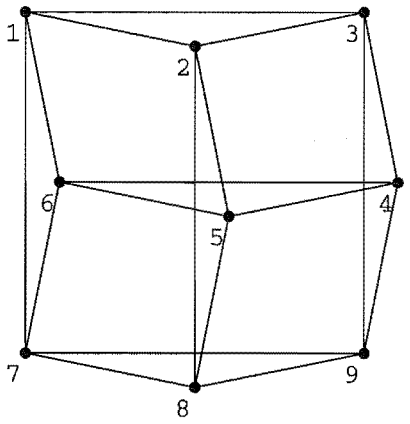
Technical Note: The concepts of reduced forms and reduction rules just presented have strong parallels to concepts in the theory of Gröbner bases, introduced in §6.14.4 *Gröbner Bases*.

Yet the question then arises: Do we ever encounter such systems as (6.14.e)? It appears to be the case that, in practice, the answer is *no*. Seemingly, whenever we have more than one linear symmetry acting, we always get more than two equations. That is, if we generate the closure of our cyclic symmetries, we get further equations. Intuitively, this means that in practice, we would have further equations; for instance, for the system (6.14.e) we would also generate something like $a+b+d==0$, etc. Let us demonstrate this on our prototypical example. If we try to find the equations for the following tensor product, we find that there are many such equations.

```
In[53]:= equations = toTensorSymbols /@ EquationsOfExpression @ expr

Out[53]= {TPS`Π2 - TPS`Π3 == TPS`Π1, TPS`Π5 - TPS`Π6 == TPS`Π4,
    TPS`Π6 - TPS`Π7 == TPS`Π1, TPS`Π5 - TPS`Π8 == TPS`Π2,
    TPS`Π4 - TPS`Π9 == TPS`Π3, TPS`Π8 - TPS`Π9 == TPS`Π7}
```

To obtain a better intuitive feel for these equations let us represent them in the form of a graph. Any two vertices in the graph will be linked if and only if they are both directly involved in some single equation in the set of governing equations.



From this diagram we can see that there is a large degree of latitude. Intuitively, the tensor product symbols have enough linkages that situations like those embodied in (6.14.e) will not arise.

How then can we make use of this abundance of linkages or connections? It is unknown to the author if the step by step term reduction algorithm will always terminate with the smallest configuration, exactly as does the Gröbner basis method. But it appears to work for a large number of examples. Thus we speculate the following.

Speculation / Conjecture / Phenomenon 6.14.A: Under appropriate conditions, the equations which arise in our systems of linear governing equations are always such that direct reduction is a canonical operation. That is, the algorithm which takes any higher order symbol and rewrites it to equivalent lower order symbols is a canonicalizing algorithm.

Evidence: We can provide some corroborating evidence for this conjecture by examining the method of direct reduction in conjunction with several different sets of governing equations. These examples are given in appendix §C.3 *Evidence for Direct Reduction Conjecture.* ■

Technical Note: Conjecture 6.14.A holds for a surprising number of cases. Yet given that there exist pseudo-counterexamples, such as (6.14.e), then it is not surprising to know that we can find corresponding pseudo-counterexamples in actual tensor settings. The next subsection gives such a pseudo-counterexample in the case when we are including auxiliary linear symmetries. The predominant issue is whether we can find a stable set of criteria whereby we can use direct reduction.

Using direct reduction — in those cases where it is a canonical operation — would greatly speed up the canonicalization of tensorial expressions with linear symmetries. The reason is that we would only have to traverse from our starting point to the set of "bottom" symbols, that is, we would be able to use a downhill method — see §C.3.4 *Reduction Structure.* Also we could cache our reductions as we find them, thus allowing us to avoid many of the redundant computations. In contrast, when using Gröbner canonicalization, we must generate the whole set of governing equations using the closure method.

The method of direct reduction has been included as an option in our overall `Canonicalize` algorithm, but in only a limited way. We still generate the whole set of governing equations. It would not be difficult to recode the algorithms in §D.13 *The Complete Canonicalization Algorithm*

to include this optimization. It will likely be highly productive to investigate statement 6.14.A further.

Shortly we will present some timings for canonicalizations involving linear symmetries, but prior to this we will discuss some of the implementation issues and the representations we will use for linear symmetry permutations. To close this subsection, note that there are other alternatives we could explore besides Gröbner canonicalization and direct reduction, but have not done so. The main idea the author envisages would be to obtain the core reduced symbols, that is the symbols which all the other symbols are reduced to, and then try different orderings on these symbols. However, these ideas are highly speculative.

6.14.8 Linear Symmetry Permutations

The previous subsections have discussed *equational* linear symmetries in detail. We have not yet actually described how to represent these by *formal* linear symmetries, the objects we actually use in our `Canonicalize` algorithm. That is the topic of this subsection.

In §6.3.1 *Permutations and Configurations* we introduced data types for both signed permutations and transpositions. We must now introduce the data type for linear symmetries.

Definition 6.14.H: A *linear symmetry* is a collection of signed permutations, all of which are the same size.

We use the following notation for linear symmetries.

```
In[54]:= Notation[⟨perms__⟩Σ==0 ⟺ LinearSymmetry[perms__]]
```

Let us give a simple example of this data type. Recall from §6.14.1 *Origins of Linear Symmetries*, the equational linear symmetry (6.14.a) of the Riemann tensor, that is

$$\mathbf{R}_{\alpha\beta\gamma\delta} + \mathbf{R}_{\alpha\gamma\delta\beta} + \mathbf{R}_{\alpha\delta\beta\gamma} = 0$$

We would represent this equational linear symmetry by the following (formal) linear symmetry.

```
In[55]:= ⟨{1, 2, 3, 4}+, {1, 3, 4, 2}+, {1, 4, 2, 3}+⟩Σ==0

Out[55]= ⟨{1, 2, 3, 4}+, {1, 3, 4, 2}+, {1, 4, 2, 3}+⟩Σ==0

In[56]:= FullForm @ %

Out[56]//FullForm=
LinearSymmetry[SignedPermutation[1, List[1, 2, 3, 4]],
   SignedPermutation[1, List[1, 3, 4, 2]],
   SignedPermutation[1, List[1, 4, 2, 3]]]
```

A linear symmetry acts on a configuration to generate an equation. It does this by acting on the configuration with each individual signed permutation of the linear symmetry, reconstituting the indices in the resulting configurations, then summing these tensors and equating this to

zero. Formally stated, the action of a linear symmetry $\langle sp_1, \ldots, sp_n \rangle_{\Sigma=0}$ on a signed configuration $t \in C$ is in essence the following.

$$\langle sp_1, \ldots, sp_n \rangle_{\Sigma=0} *_{equation} t \longrightarrow \left(0 == \sum_{i=1}^{n} reconstitute\,(sp_i *_c t) \right) \tag{6.14.f}$$

Technical Note: In our description of how a linear symmetry is applied to a configuration to yield an equation, (6.14.f), we used $*_{equation}$ and ***reconstitute*** (...). Technically, the action $*_{equation}$ does not exist. In actuality, the code uses the routine `buildEquation`. Moreover, the configurations are not reconstituted since the complete algorithm works with the linear symmetries in terms of cached configurations — see §D.13 *The Complete Canonicalization Algorithm*.

The correspondence between the linear symmetry for `R`, $\langle \{1, 2, 3, 4\}_+, \{1, 3, 4, 2\}_+, \{1, 4, 2, 3\}_+ \rangle_{\Sigma=0}$, and its equational form $\mathbf{R}_{\alpha\beta\gamma\delta} + \mathbf{R}_{\alpha\gamma\delta\beta} + \mathbf{R}_{\alpha\delta\beta\gamma} = 0$ should now be clear.

Just as signed permutations are usually associated with generators for the symmetries of a particular tensor, so also, linear symmetries are usually associated with a given tensor. In §6.7.2 *Specification of Generators*, we gave several examples using the function `DeclareSymmetries`. This function declared the primitive symmetries of a given tensor to be those specified in the declaration. Let us include the above linear symmetry in the declared set of primitive symmetries for the Riemann tensor.

```
In[57]:= DeclareSymmetries[R, 4, {(1 ↔ 2)_, (3 ↔ 4)_, {3, 4, 1, 2}_+,
            ⟨{1, 2, 3, 4}_+, {1, 3, 4, 2}_+, {1, 4, 2, 3}_+⟩_Σ==0}]
```

Now `Canonicalize` will use the specified linear symmetry when canonicalizing Riemann tensors. (Actually, this symmetry is one of the default symmetries automatically loaded by the package.)

There are other equations that can be added besides just standard linear symmetries. For instance, there exists a relationship involving the Riemann tensor and the swapping of any two covariant derivatives — see [81, 240, 255, 326, 333]. The *Tensors* package has a facility to add such "auxiliary" equations to the set of governing equations generated by the linear symmetries. This function is `AddAuxiliaryEquations`. Here is an example involving the double covariant derivative of the four-vector potential.

```
In[58]:= AddAuxiliaryEquations[{
            A_{α;μν} - A_{α;νμ} == R^β_{αμν} A_β,
            A_{μ;αν} - A_{μ;να} == R^β_{μαν} A_β,
            A_{ν;μα} - A_{ν;αμ} == R^β_{νμα} A_β}]
```

Now the governing set of equations includes those reachable from the linear symmetries and the auxiliary equations.

```
In[59]:= EquationsOfExpression[A^β R_{βαμν}]
```

```
Out[59]= {A      - A      == A^β R      , -A^β R      + A^β R      == A^β R      ,
          α ; μ ν    α ; ν μ          β α μ ν          α μ β ν          β μ α ν          β α μ ν

          A      - A      == A^β R      , -A      + A      == -A^β R      }
           μ ; α ν    μ ; ν α          β μ α ν          ν ; α μ    ν ; μ α          α μ β ν
```

`Canonicalize` makes use of the full set of governing equations.

```
In[60]:= Canonicalize[A      - A      + A^β R      - A^β R      - R^β      A  + A      ]
                       μ ; ν α    ν ; α μ          β μ α ν          α μ β ν          ν μ α  β    α ; μ ν

Out[60]= A      + A      - A
           α ; μ ν    μ ; α ν    ν ; α μ
```

Just for comparison, it is reassuring to also perform this canonicalization using direct reduction as opposed to Gröbner canonicalization.

```
In[61]:= Canonicalize[A      - A      + A^β R      - A^β R      - R^β      A  + A      ,
                       μ ; ν α    ν ; α μ          β μ α ν          α μ β ν          ν μ α  β    α ; μ ν
           LinearSymmetryMethod → DirectReduction]

Out[61]= A      + A      - A
           α ; μ ν    μ ; α ν    ν ; α μ
```

The results are the same, as they should be. The last example brings up an interesting dilema. By suitable inspection one can concoct the following example, where Gröbner canonicalization works yet direct reduction does not.

```
In[62]:= Canonicalize[A      - A      - A      + A      - A      + A      ,
                       α ; μ ν    α ; ν μ    μ ; α ν    μ ; ν α    ν ; μ α    ν ; α μ
           LinearSymmetryMethod → DirectReduction]

Out[62]= A      - A      - A      + A      + A      - A
           α ; μ ν    α ; ν μ    μ ; α ν    μ ; ν α    ν ; α μ    ν ; μ α

In[63]:= Canonicalize[A      - A      - A      + A      - A      + A      ,
                       α ; μ ν    α ; ν μ    μ ; α ν    μ ; ν α    ν ; μ α    ν ; α μ
           LinearSymmetryMethod → GröbnerBases]

Out[63]= 0
```

At first sight, the above disparity is rather disturbing, since it seems to provide a contradiction to our Conjecture 6.14.A. Yet, strictly speaking, there is no contradiction since our conjecture concerns only "proper" linear symmetries, whereas the above example has entered the realm of "auxiliary" linear symmetries.

In spite of the qualification, I still feel that Conjecture 6.14.A should be stated as a "soft" conjecture. This is because I believe that in all likelihood, an unqualified version of it will be broken somehow. This is a matter of personal experience and some gut instinct. I somehow feel that it cannot hold in general, especially since we can concoct pseudo-counterexamples by considering "auxiliary" linear symmetries. Thus more work is needed in exploring the "appropriate conditions" and limits of Conjecture 6.14.A. However, for the cases when it is true, it could lead to dramatic improvements in canonicalization times.

Technical Note: Actually in reference to our pseudo-counterexample above, if we included the linear symmetry $\mathbf{A}_{\alpha\,;\,\mu\,\nu} - \mathbf{A}_{\alpha\,;\,\mu\,\nu} - \mathbf{A}_{\alpha\,;\,\nu\,\mu} - \mathbf{A}_{\mu\,;\,\alpha\,\nu} + \mathbf{A}_{\mu\,;\,\nu\,\alpha} - \mathbf{A}_{\nu\,;\,\mu\,\alpha} + \mathbf{A}_{\nu\,;\,\alpha\,\mu} == 0$ then direct reduction would indeed yield the same answer as does Gröbner canonicalization. However, Gröbner canonicalization, in contrast to direct reduction, does not need the full set of linear symmetries to be explicitly stated, as was evidenced by our example.

To close this section, we should mention that `DeclareSymmetries` can be called with a triple for the valence of the tensor as opposed to just a single number of indices. Specifically, `DeclareSymmetries[`*T*`, {`*n*`, `*cd*`, `*pd*`}, `*symmetries*`]` will declare the symmetries *symmetries* for the primitive tensor denoted *T* with *n* normal indices, *cd* covariant derivatives, and *pd* partial derivatives. The valence-triple can contain patterns. For instance, the Bianchi identities (6.14.b) can be added as follows.

```
In[64]:= DeclareSymmetries[R, {4, cd_ /; cd ≥ 1, _},
           {(1 ↔ 2)_, (3 ↔ 4)_, {3, 4, 1, 2}_+,
             ⟨{1, 2, 3, 4}_+, {1, 3, 4, 2}_+, {1, 4, 2, 3}_+⟩_{Σ==0},
             ⟨{1, 2, 3, 4, 5}_+, {1, 2, 4, 5, 3}_+, {1, 2, 5, 3, 4}_+⟩_{Σ==0}}]
```

Consequently, some expressions can take a significant length of time to canonicalize since the set of governing equations is so large. For instance note the number of governing equations alone for the following tensor product.

```
In[65]:= EquationsOfExpression[R^{α δ β μ}_{; ν} R_{α β δ τ ; ε} R^{τ γ}_{μ γ}]_{len}

Out[65]= 66
```

6.14.9 Implementation Of Linear Symmetries

We will not cover any of the other implementation details involving linear symmetries in any depth. The interested reader should consult the code section §D.13 *The Complete Canonicalization Algorithm* for details. Obviously the overall method should be clear. However, we will briefly mention some details of the process beyond the background given in the previous subsections of this section. In no particular order: (*i*) we must handle the various options passed to the canonicalization algorithm; (*ii*) we must account for the canonicalization of the free indices at each stage of the linear symmetry generation; (*iii*) we must cache all of the computations, and associate these caches with the appropriate tensor symbols to avoid recomputation; (*iv*) we must create code for both `AdditionalEquations` and `EquationsOfExpression`; (*v*) we must include code for clearing the tensor caches; (*vi*) if the free indices are canonicalized we can use a more efficient ordering by skipping some of the comparisons; (*vii*) we must implement reduction methods for both direct reduction and Gröbner canonicalization.

The following is the generalization of Theorem 6.7.A.

Theorem 6.14.C: The function `Canonicalize`, implemented in §D.13 *The Complete Canonicalization Algorithm*, is a canonicalizing function on tensor products.

Proof: Omitted. (It is similar to Theorem 6.7.A.) ■

Let us finally observe some results and timings for our overall `Canonicalize` algorithm. This is the subject of the next subsection.

6.14.10 Examples and Timings of Canonicalizations with Linear Symmetries

Because of the linear symmetries of R, some tensor products can change shape.

```
In[66]:= Canonicalize[R^{β τ α λ} R^{ν γ}_{λ α} T^{μ}_{ν γ}] // Timing
```

$$\text{Out[66]= } \left\{0.566667\ \text{Second},\ -\frac{1}{2}\,\mathbf{R}^{\beta\,\tau\,\alpha\,\gamma}\,\mathbf{R}^{\delta\,\epsilon}_{\alpha\,\gamma}\,\mathbf{T}^{\mu}_{\delta\,\epsilon}\right\}$$

When we set our linear symmetry method to none then our overall `Canonicalize` algorithm ignores linear symmetries, and so in essence, defaults to the optimized canonicalize algorithm presented in §6.11 *The Optimized Algorithm*. Let us demonstrate this.

```
In[67]:= Canonicalize[R^{β λ γ τ}_{;ν} + R^{β λ γ τ}_{ν ;} + R^{β λ τ γ}_{ν ;},
           LinearSymmetryMethod → None]
```

$$\text{Out[67]= } \mathbf{R}^{\beta\,\lambda\,\gamma\,\tau}_{\quad;\,\nu} - \mathbf{R}^{\beta\,\lambda\,\gamma\quad\tau}_{\quad\nu\,;} + \mathbf{R}^{\beta\,\lambda\,\tau\quad\gamma}_{\quad\nu\,;}$$

However if we include linear symmetries, then because of the Bianchi identities (6.14.b), the above expression is in fact equivalent to 0.

```
In[68]:= Canonicalize[%]
Out[68]= 0
```

Actually, it is comforting to verify that the same result is returned if we use direct reduction as opposed to Gröbner canonicalization.

```
In[69]:= Canonicalize[%%, LinearSymmetryMethod → DirectReduction]
Out[69]= 0
```

For another example here is the timing for the following canonicalization.

```
In[70]:= Canonicalize[R^{α β λ}_{α} R^{τ γ}_{β λ ;ν} T^{μ}_{τ γ}] // Timing
```

$$\text{Out[70]= } \left\{1.13333\ \text{Second},\ \mathbf{R}^{\alpha\,\beta\,\gamma}_{\quad\alpha}\,\mathbf{R}^{\delta\quad\epsilon}_{\beta\quad\gamma\,;\,\nu}\,\mathbf{T}^{\mu}_{\delta\,\epsilon}\right\}$$

Let us again compare Gröbner canonicalization to direct reduction but this time include the timing information

```
In[71]:= ClearTensorCaches[]

In[72]:= Canonicalize[R^{α β λ}_{α R^{τ γ}_{β λ ; ν T^{μ}_{τ γ},
              LinearSymmetryMethod → DirectReduction] // Timing

Out[72]= {1.1 Second, R^{α β γ}_{α R^{δ ε}_{β γ ; ν T^{μ}_{δ ε}}}}
```

We cleared the tensor caches above in order to get an accurate timing for the canonicalization. If we had not done so, then the cached results would have been used. For instance, if we now canonicalize the expression, the answer will be given extremely quickly.

```
In[73]:= Canonicalize[R^{α β λ}_{α R^{τ γ}_{β λ ; ν T^{μ}_{τ γ}] // Timing

Out[73]= {0.0833333 Second, R^{α β γ}_{α R^{δ ε}_{β γ ; ν T^{μ}_{δ ε}}}}
```

Finally to get an idea of just how many governing equations can arise, note that the following tensorial expression gives rise to some 285 different equations. (We need to increase the recursion limit since there are so many governing equations.)

```
In[74]:= $RecursionLimit = 512;

In[75]:= Canonicalize[R^{α δ β μ}_{; ν R_{α β δ τ ; ε R^{τ γ ε}_{μ ; γ}] // Timing

Out[75]= {121.967 Second,
           R^{α β γ δ ε}_{; R^{ζ η}_{α γ ; ζ R_{β ε δ η ; ν - 1/4 R^{α β γ δ ε}_{; R^{ζ η}_{α β ; ζ R_{γ δ ε η ; ν}}

In[76]:= EquationsOfExpression[R^{α δ β μ}_{; ν R_{α β δ τ ; ε R^{τ γ ε}_{μ ; γ}]_len

Out[76]= 285
```

This yet again reinforces the issue that using direct reduction could potentially save a large amount of superfluous computations. Once the equations are cached, it is sometimes much faster to perform direct reduction than it is to perform Gröbner canonicalize.

```
In[77]:= Canonicalize[R^{α δ β μ}_{; ν R_{α β δ τ ; ε R^{τ γ ε}_{μ ; γ}] // Timing

Out[77]= {55.8 Second,
           R^{α β γ δ ε}_{; R^{ζ η}_{α γ ; ζ R_{β ε δ η ; ν - 1/4 R^{α β γ δ ε}_{; R^{ζ η}_{α β ; ζ R_{γ δ ε η ; ν}}

In[78]:= Canonicalize[R^{α δ β μ}_{; ν R_{α β δ τ ; ε R^{τ γ ε}_{μ ; γ},
              LinearSymmetryMethod → DirectReduction] // Timing
```

```
Out[78]= {2.8 Second,
```

$$\mathbf{R}^{\alpha\beta\gamma\delta\ \epsilon\ \zeta\ \eta}_{;\ \ \ \

The simple closure algorithm for generating all reachable configurations from a starting configuration was first given in §6.5 *Generating Configurations*. Formally, we presented an algorithm for calculating the orbit of configurations reachable from an initial configuration under our permutation and relabeling group action. Variants of this basic closure algorithm were also used in §6.9.4 *Transpositional Canonicalization and* GenerateConfigurations$_T$, §C.4 *Canonicalizing in Stages*, and §D.13 *The Complete Canonicalization Algorithm*. Besides proving the "correctness" of the algorithm `generateConfigurations`, we commented on some basic algorithms in computational group theory which share similarities to our algorithm. The main reason why standard computational group theory is not directly applicable, is that our permutation and relabeling action entangles the group operations.

The next facet of our overall algorithm was to introduce an ordering on signed configurations in §6.6 *The Ordering of Configurations*. The ordering was chosen in such a way that it is almost a total ordering (formally, a linear quasi-ordering). With this ordering we can define the notation of a configurational minimum or minima. The case when there is more than one minimum only arises when the tensor under consideration is identically zero. In the case when there is only one minimum we choose this configuration to be our canonical configuration.

We put the concepts of the preceding sections together to form the basic canonicalization algorithm in §6.7 *The Basic Canonicalization Algorithm*. Briefly the algorithm consists of (*i*) encoding a tensor product into a corresponding signed configuration, (*ii*) finding all the permutation generators prescribed by the primitive symmetries of the tensors in the tensor product, (*iii*) generating all equivalent configurations under the symmetries, (*iv*) picking the minimum equivalent configuration from the generated set, and finally (*v*) reconstituting the configurations back to a tensor product. Also, in this section we formally defined what "canonical" meant and then proved that our basic algorithm was a canonicalizing function.

The fundamental ideas involving group theory were presented in §6.8 *Generators and Group Theoretic Underpinnings*. We split the set of symmetry generators into complex, degenerate, and transpositional symmetries. We used the computation group theory package GAP to verify and illustrate several key concepts and ideas. We then proceeded to formally introduce the concept of a Jointly Recursively Directional and Extrema Stabilizing (JRDES) set. The JRDES concept is close to a notion used in computational group theory, that of a base with a strong generating set; only in the case of JRDES, the base is in a sense "flexible". We showed that sets of adjacent transpositions were JRDES, gave a simple coding of an algorithm to determine whether a given set of generators was JRDES, and proved several results using JRDES generating sets.

Once we could formally qualify what was required of our generating sets, we introduced the notation of transpositional canonicalization in §6.9 *Transpositional Canonicalization*. This is essentially mini-canonicalization but only across the transpositional symmetries of the underlying tensor product. By using the reducing swap operator, $\leftrightarrows_?$, we could find a transpositionally minimum configuration extremely efficiently. We proved that the reducing swap operator always predicted when it was best to perform a swap. However, for our transpositional canonicalization algorithm to work, we needed to conjecture that we could obtain the transpositionally minimum configuration in a "downhill fashion". Although a direct proof has not been found by the author, hard evidence that the conjecture is true is given in §C.2 *Evidence for Steepest Descent Conjecture*. The section concludes with a proof that when

transpositional canonicalization is included in our algorithm for the generation of configurations, we will obtain a configuration from each of the reachable transpositional equivalence classes.

The question of whether a tensor is identically zero or not is covered in §6.10 *Identically Zero Tensors*. At first the question is addressed with respect to the basic canonicalization algorithm. But once we include transpositional canonicalization into our algorithm, the theoretical question becomes much more difficult to answer. Fortunately, the answer is still extremely easy to discern in practice. The concept of an induced transposition played a pivotal role in regards to the issue of "identically zero".

Finally in §6.11 *The Optimized Algorithm*, we brought together the developments of the previous subsections to yield a much more efficient algorithm. Loosely, we included (*i*) the canonicalization of the free indices, the theory of which was developed in §6.8 *Generators and Group Theoretic Underpinnings*; (*ii*) the removal of superfluous permutation generators by stabilizing on the canonicalized free indices; (*iii*) we included transpositional canonicalization into the generation of equivalent configurations; and finally (*iv*) we incorporated our enhancements to the routines for determining whether a tensor was identically zero or not. We demonstrated that the optimized algorithm was vastly superior to the basic canonicalization algorithm. This efficiency occurs in both storage and in execution time.

Due to the presence of partial derivatives, in certain circumstances indices are not allowed to be raised and lowered in a complementary way by the metric. The reasoning behind this was given in §6.12 *Refinements for Partial Derivatives*. To allow for this restriction we had to specialize the data structures for summed indices, by including fixed elevations into the summed indices themselves. This necessitated corresponding changes to the ordering on configurations, transpositional canonicalization, and the determination of when a tensor is identically zero. The changes to transpositional canonicalization were somewhat involved. Yet in the end, all of our previous results still held.

In practice, tensors with indices from different classes are used, and thus it was all but necessary to extend our algorithm yet again. This was the topic of §6.13 *Refinements for Mixed Index Classes*. As with fixed elevations, the basic data structure of a summed index was changed by affixing an index class to each summed index. As before, incorporating this extension forced corresponding changes to the ordering on configurations, transpositional canonicalization, and the determination of when a tensor is identically zero. Fortunately, all of our key concepts were correspondingly extensible.

The final section, §6.14 *Linear Symmetries and the Complete Algorithm*, introduced the concept of a linear symmetry and its attendant consequences. Several examples from physics were given to motivate such symmetries. The various possibilities for a canonical function over linear symmetries were discussed before presenting some brief background on the theory of Gröbner bases. We utilized the properties of Gröbner bases in order to create a function that canonicalized across linear symmetries. We used the natural ordering on configurations to obtain a derived term ordering for our Gröbner canonicalization. Similar to concepts in Gröbner basis theory, we introduced direct reduction. All of these developments, together with almost all of the concepts and their changes and variants introduced in the previous sections, combine and culminate in the routine `Canonicalize`. We closed the section by presenting

some of the auxiliary functions of the *Tensors* package and finally several examples of the algorithm working in practice.

6.15.2 Lingering Issues

Before we conclude this chapter let us briefly comment on several outstanding issues. First, the code is not perfect in that there are some small optimizations we could have possibly made, yet have elected not to implement. For instance, our ordering, $\prec_c$, always tries to compare two signed configurations according to their index classes in a last ditch effort to find an ordering on the configurations. However, if we are not including index classes explicitly in the indices, than this is clearly a waste of time. Note, though, that this case will arise very rarely in practice, since for $c \in C$ there is only one other configuration that will fall through to this case, $-c$. Although this might seem like an excruciatingly small point, we could probably attain a further 10% efficiency in our algorithm by performing such optimizations.

However, such minute detail is not warranted in our approach. If we really desire high speed, it would be much more productive to write the core algorithm in, say, C++ and call it through *MathLink*[342, 343]. This would likely yield an improvement of two orders of magnitude or more. Also, any of the other potential optimizations given in the previous subsections could be investigated. We further comment on this in the next subsection.

We could have included an option for a metric connection but have decided against this. If anyone needs to use metric connections, it is eminently possible to modify the encode tensors algorithm so it keeps fixed elevations for indices in tensors with covariant derivatives.

In every algorithm but the basic one, an early step is to canonicalize all of the free indices. After this, every free index will remain in a fixed position. Most of the work is then just dealing with different configurations where the summed indices change positions. Therefore, we can introduce partial versions of the comparison operators defined in §1.6.2. The complete algorithm does exactly this. And consequently, the complete algorithm is slightly faster than the optimized algorithm. Actually, a partial comparison would be pretty easy to add to the optimized algorithm, but it has not been done.

In accordance with many books using tensors, we allow the symbol 'δ' to be an index. However, the user should be extremely careful since it is almost always used as the delta function as well. The symbols that we use for tensor indices are protected. This stops them being assigned values, which would seriously compromise computations. We could have attempted to define a context sensitive notation that parsed the indices into some non-default context and also formatted these non-default indices back as standard indices. Such an approach would most likely have been problematic, and hence we have avoided implementing it.

As before, we need to point out that we need our generating sets for our tensor products to be JRDES in order that our more advanced algorithms function correctly. If a primitive tensor has non-adjacent transpositions, then we either need to enter these transpositions as signed permutations, or we need to introduce an intermediate tensor whose index symmetries are a

permuted version of the originals, a version in which the transpositional symmetries are adjacent.

Currently, the complete algorithm needs to recanonicalize the free indices at each stage when taking linear symmetries into account. At a gut level, I feel that it may be possible to devise some way to canonicalize the free indices — once and for all — when dealing with linear symmetries. However, this has not been investigated in any depth.

Since the results are cached at each stage, it may be necessary in extremely large scale calculations to clear the tensor caches.

Some of the major speed enhancements made in our algorithms depend quite strongly on conjectures. Even though we can verify these conjectures by brute force up to a certain order and by random sampling beyond that, it would still be highly desirable to find proofs for these conjectures. During the course of my research, I have come to suspect the following is possible to prove. Loosely, after the free indices have been canonicalized, if we should transpositionally swap indices a_i and a_j to make a configuration smaller and if i and j are not adjacent, then there exists another index between them, say k such that the configuration would be made smaller by either swapping i with k or swapping k with j. It should be possible to prove this by logical inference, but the inference engine of *Mathematica* has so far proved insufficient. I have developed several extensions to the inference engine which would enable one to prove logical results more effectively. Yet even these are still insufficient to the task. Possibly some theorem provers would be more suitable. And this is yet another area for further research.

6.15.3 Concluding Remarks

This subsection concludes the presentation of the algorithm for canonicalizing tensorial expressions. In the next chapter we progress on to some applications using the `Canonicalize` algorithm. Indeed, in the rest of this thesis, the `Canonicalize` algorithm will be used only in the capacity as a tool.

It should be abundantly clear that the algorithmic advances made in this chapter are extremely significant. It appears that we have discovered an algorithm that is orders of magnitude better than all others known to the author. It has been implemented in its various variants and is eminently usable on a practical basis.

The algorithm embodied in `Canonicalize` handles all of the variants developed throughout this chapter. It handles dummy indices, with both natural elevations and fixed elevations. It easily canonicalizes expressions with mixed index classes. It handles linear symmetries either by Gröbner canonicalization or by direct reduction. The results are cached so that any other equivalent tensor (up to free indices) is handled extremely quickly. It allows additional linear symmetries to be added to the set of overall governing equations. And finally, comparatively speaking, it is extremely efficient.

It is important to note that even though the canonicalization algorithm depends on a conjecture, it will *never* give a wrong answer. It is possibly conceivable that it may give an

answer that is not in the most canonical form, but it will *never* be wrong. That is, any answer calculated with the algorithm will always be correct, just possibly not as simple as it could be.

Technical Note: Of course we make the above statement with the proviso that we are assuming the underlying machine does not have hardware faults and / or *Mathematica* does not have core defects in its computational engine.

It would be nice to somehow prove the transpositional canonicalization conjectures, that is, Conjecture 6.9.A, and its extensions to Conjecture 6.12.A and Conjecture 6.13.B. Yet, proofs of these conjectures are not strictly necessary since §C.2 *Evidence for Steepest Descent Conjecture* gives extremely strong evidence for their truth-hood.

For the future, in order to further increase the speed of the algorithm, there are several factors we might pursue. These factors have been mentioned throughout the chapter. First and most obvious is to code the algorithm in C++ or an equivalent low-level compiled language. This would probably speed up the algorithm between 50-500 times. It should be clear that our canonicalization algorithms are easily translated to a language like C++ since they do not "intrinsically" use any of the "higher level" programming structure of *Mathematica*. This is, however, not the case for hueristic algorithms such as Ricci[202] or MathTensor[258]. It would be excedingly difficult to rewrite something that fundamentally depends on the pattern matching engine since to replicate this in C++ is quite hard.

Next in importance in terms of efficiency would probably be adapting our algorithms for canonicalization in stages and proving their "correctness"— see §C.4 *Canonicalizing in Stages*. Next after this would probably be the further investigations of Conjecture 6.14.A, trying to put hard limits on when it will hold.

It should again be pointed out that the method of direct reduction holds vast promise. The results in §6.14.10 *Examples and Timings of Canonicalizations with Linear Symmetries* demonstrate that direct reduction has the potential to be *orders of magnitude faster* than Gröbner canonicalization. Direct reduction should be further investigated.

In summary, the original developments in this chapter have yielded a highly efficient algorithm to provably canonicalize indexed tensorial objects. To the best of the author's knowledge, this algorithm is orders of magnitude faster than comparable algorithms.

Chapter 7
Tensor Calculus, Applications, and Quasi-Spin

7.1 Introduction

7.1.1 Overview

Throughout the previous chapters we have developed notations, language modifications, generic prototypes, and finally an algorithm for the canonicalization of tensorial expressions. In this chapter we combine aspects from all of the previous chapters by presenting a system for tensorial calculus.

In this chapter we can use our notations, our language modifications, and our canonicalization routines in order to perform calculations with tensors. Most of the calculations in this section are examples from general relativity. They are intended to give a guide to the sorts of calculations one would normally perform and the ease with which they can be structured. As mentioned in the introduction, there are several systems available that perform computations in a general relativity setting. The goals of our implementation are to have as intuitive an interface as possible, while at the same time to have the most efficient general canonicalizing algorithm, which can be extended in many ways. As such, we develop a general indicial manipulation package according to the classification of [150]. Further specialization to specific calculations is easily achieved using the framework presented herein.

We start with a discussion of indices, coordinates, and conventions used. We then present the notations for tensorial assignment. Succeeding this are some basic definitions for common tensors in general relativity, for instance, the Ricci and Riemann tensors, the metric tensor, the Weyl tensor, etc. Some of the basic handling methods for tensors and tensor calculus are then presented. The handling methods use the language modifications of the earlier sections.

Since the MathTensor system of Parker & Christensen[258] is widely known and used, it provides a convenient benchmark against which we can compare the *Tensors* package. In this regard, some of the examples we illustrate are similar to or the same as those of the MathTensor documentation. In this way, the user will be able to compare and contrast the differing styles of notation, functionality, and elegance. There are several other sources of examples upon which this section is based. None of the examples in this section constitute anything fundamentally

new from the point of view of applications, except the result in quasi-spin, which is not fully reported here. Rather, it is the method of performing the computations as well as our canonicalization algorithm itself which are original. To again reiterate, to the best of the author's knowledge, our canonicalization algorithm is orders of magnitude faster than other competing canonicalization algorithms on larger scale problems. Occasionally, we will comment on the speed of other algorithms.

As a preliminary to this chapter, let us load the *Tensors* package and the *Prototypes* package.

```
In[1]:= << Tensors`
```

```
In[2]:= << Prototypes`
```

The *Tensors* package loads the *Notation* package, if it is not already loaded. Similarly, the *Prototypes* package loads the *Assign* package, if that is not already loaded.

Let us also load the common notations.

```
In[3]:= << CommonNotations`
```

We next proceed on to consider the manipulation of tensors and the introduction of a calculus for our tensors

7.2 Tensor Manipulations and Tensor Calculus

7.2.1 Indices And Coordinates

The whole subject of §6.13 *Refinements for Mixed Index Classes* concerned the modifications to our canonicalization algorithm necessary to handle mixed index classes. However, it was never actually stated what constitutes an index or how to specify the coordinates that an index can take on. In this subsection, we present the commands for declaring coordinates and indices.

Loading the *Tensors* package has the effect, amongst many others, of adding the following functions.

`DeclareIndexClass[` *class*, $\{i_1, i_2, \ldots, i_n\}$`]`	declare that the indices $i_1 \ldots i_n$ are the base indices for the index class *class*
`ClassOfIndex[`*index*`]`	return the class which *index* belongs to
`ValidIndexClasses`	return all the index classes in use

`ValidIndexClassQ[` *class*`]`	determine whether *class* is a valid index class
`IndexQ[` *index*`]`	determine whether *index* is an index of any of the valid index classes
`BaseIndices[` *class*`]`	return all the base or root indices which belong to the index class *class*

The functions for declaring and querying indices.

Upon loading, the *Tensors* package automatically declares two sets of indices: `GeneralIndices` and `SpaceTimeIndices`. This is in accordance with the conventions that most authors adopt. Inside the *Tensors* package, this is accomplished by performing the following commands.

```
In[1]:= DeclareIndexClass [ SpaceTimeIndices ,
          {α, β, γ, δ, ϵ, ζ, η, λ, μ, ν, ξ, σ, τ, χ, ω}]
        DeclareIndexClass [ GeneralIndices ,
          {a, b, c, d, i, j, k, l, m, n, o, p, q}]
```

The function `DeclareIndexClass` and its related functions `ClassOfIndex`, `IndexQ`, `ValidIndexClasses`, and `BaseIndices`, are all reasonably straightforward to implement. Currently, the tensor and canonicalization algorithms have been restricted to use only symbols. The interested reader can consult the detailed code in §E.3 *Set Up Index Conventions*.

The base indices of a given class of indices give only a finite number of acceptable indices. However, during the course of a calculation, further indices of the same class may be needed. For this reason, any base index suffixed with a positive integer is also treated as an index of the same class. That is, if x is a base index, then x1, x2, ... are also acceptable indices. Moreover, x$1, x$2, etc., are also acceptable.

Under most circumstances, it is not critical which class of indices one uses for which function. Some particular algorithms the user might write can depend on one class or another; but other than keeping track of such indices, no intrinsic requirements exist as to the particular class used.

To provide values over which the indices of a particular class can range, the *Tensors* package also declares corresponding coordinates.

`DeclareCoordinates[` $\{c_1, c_2, \ldots, c_n\}$`]`	declare that the c_i are to be treated as valid coordinates. Note: a coordinate cannot be an index
`CoordinateQ[`*coord*`]`	determine whether *coord* is a coordinate
`DeclareCoordinateClass[` *class*, $\{c_1, c_2, \ldots, c_m\}$`]`	declare that the coordinates of each index in the index class *class* range over the coordinates $c_1, \ldots, c_m$

The functions for declaring and querying coordinates.

Upon loading, the *Tensors* package declares a "base" list of objects which should be treated as coordinates.

```
In[3]:= DeclareCoordinates[{0, 1, 2, 3, t, x, y, z, r, ρ, Θ, ϕ}]
```

To reiterate, coordinates are distinct from indices. Coordinates provide the values over which indices can range, and over which dummy indices are to be summed. In order to carry out

expansions over sums, the default coordinates for each index class must be declared. The *Tensors* package makes the following choices as to default ranges of coordinates.

```
In[4]:= DeclareCoordinateClass[SpaceTimeIndices, {0, 1, 2, 3}];
        DeclareCoordinateClass[GeneralIndices, {1, 2, 3}];
```

The function `ExpandContraction` will expand a tensorial expression containing a dummy index into a sum of terms according to the class of the dummy index. Here is an expansion over a general index.

```
In[6]:= T^i T_i // ExpandContraction
```

$$\text{Out[6]= } \mathbf{T}^{1}\,\mathbf{T}_{1} + \mathbf{T}^{2}\,\mathbf{T}_{2} + \mathbf{T}^{3}\,\mathbf{T}_{3}$$

Similarly, when the dummy index is a space-time index, then it is expanded according to the default indices of that class.

```
In[7]:= T^α T_α // ExpandContraction
```

$$\text{Out[7]= } \mathbf{T}^{0}\,\mathbf{T}_{0} + \mathbf{T}^{1}\,\mathbf{T}_{1} + \mathbf{T}^{2}\,\mathbf{T}_{2} + \mathbf{T}^{3}\,\mathbf{T}_{3}$$

If we change the default indices of a class, then the summation occurs over the new coordinates.

```
In[8]:= DeclareCoordinateClass[SpaceTimeIndices, {t, ρ, θ, ϕ}];

In[9]:= T^α T_α // ExpandContraction
```

$$\text{Out[9]= } \mathbf{T}^{t}\,\mathbf{T}_{t} + \mathbf{T}^{\theta}\,\mathbf{T}_{\theta} + \mathbf{T}^{\rho}\,\mathbf{T}_{\rho} + \mathbf{T}^{\phi}\,\mathbf{T}_{\phi}$$

It is trivial to introduce a new class of indices or new coordinates. For instance, let us introduce a set of indices for the structure constants of $SU(2)$ for use in some calculations involving Yang-Mills fields. This problem is also treated by Parker & Christensen[258]. For further information on group theory in physics, consult Butler[45], and for texts treating Yang-Mills gauge fields, consult [138, 186, 278]. We will use gothic i and j indices for these $SU(2)$ structure indices.

```
In[10]:= DeclareIndexClass[SU2StructureIndices, {i, j}];
```

In the case of $SU(2)$, the structure indices run from 1 to 3, since there are three gauge fields. (To consider $SU(3)$, which is used in the color symmetries of the standard model, we would extend our set of declared coordinates by including 5, 6, 7 and 8 since there are eight gauge fields.)

```
In[11]:= DeclareCoordinateClass[SU2StructureIndices, {1, 2, 3}];
```

As an illustration, let us expand over the contracted indices in the following term, which is present in the Yang-Mills Lagrangian.

```
In[12]:= ExpandContraction[F^i_αβ F_i^αβ] // Short
```

Out[12]//Short=

$$\mathbf{F}^{1}{}_{t\,t}\,\mathbf{F}_{1}^{t\,t} + \mathbf{F}^{1}{}_{t\,\theta}\,\mathbf{F}_{1}^{t\,\theta} + \mathbf{F}^{1}{}_{t\,\rho}\,\mathbf{F}_{1}^{t\,\rho} + \mathbf{F}^{1}{}_{t\,\phi}\,\mathbf{F}_{1}^{t\,\phi} + \mathbf{F}^{1}{}_{\theta\,t}\,\mathbf{F}_{1}^{\theta\,t} + \mathbf{F}^{1}{}_{\theta\,\theta}\,\mathbf{F}_{1}^{\theta\,\theta} + \langle\langle 37 \rangle\rangle + \mathbf{F}^{3}{}_{\rho\,\phi}\,\mathbf{F}_{3}^{\rho\,\phi} + \mathbf{F}^{3}{}_{\phi\,t}\,\mathbf{F}_{3}^{\phi\,t} + \mathbf{F}^{3}{}_{\phi\,\theta}\,\mathbf{F}_{3}^{\phi\,\theta} + \mathbf{F}^{3}{}_{\phi\,\rho}\,\mathbf{F}_{3}^{\phi\,\rho} + \mathbf{F}^{3}{}_{\phi\,\phi}\,\mathbf{F}_{3}^{\phi\,\phi}$$

It is evident that the indices of the two classes were handled as dictated by the declared coordinates of their respective index classes. Actually, since it is more common to work with indices ranging from 0 to 4, let us revert back to this set of default coordinates for the space-time indices.

```
In[13]:= DeclareCoordinateClass[SpaceTimeIndices, {0, 1, 2, 3}];
```

It should be noted that one can work in any class of indices (as long as they are symbols) and over any coordinates.

7.2.2 Dummy Indices Revisited and Reindexing

As was first mentioned in §3.5.3 *Dummy Indices*, reindexing expressions is unfortunately an intrinsic action that cannot readily be avoided. Under the current working paradigm, it is not possible, at least in the author's opinion, to keep track of used indices without major modifications to the basic calculation mechanisms of the underlying *Mathematica* language. Indeed, this appears to also be generally true for the other computer algebra systems designed to handle tensors, such as MathTensor[258], Ricci[202], etc. The reason is fundamental to the Einstein summation convention itself. It states that if an index appears twice in an expression, then it is summed over. The dilemma originates in what constitutes a "complete expression". For instance, consider

```
In[14]:= expr = T^i T_i
```

Out[14]= $\mathbf{T}^{i}\,\mathbf{T}_{i}$

Then we might think of `expr` as a "complete expression". However, a user might then request `expr`^{2}, in which case `expr` is no longer a complete expression but only part of a greater whole, and as such, the Einstein summation convention is broken by normal multiplication.

```
In[15]:= expr * expr
```

Out[15]= $\left(\mathbf{T}^{i}\right)^{2}\left(\mathbf{T}_{i}\right)^{2}$

This follows because `expr * expr` $\longrightarrow \mathbf{T}^{i}\,\mathbf{T}_{i}\,\mathbf{T}^{i}\,\mathbf{T}_{i} \longrightarrow \left(\mathbf{T}^{i}\right)^{2}\left(\mathbf{T}_{i}\right)^{2}$. This is mathematical nonsense. To avoid this erroneous behavior, we need at least one pair of dummy indices to be automatically renamed. If we use normal *Mathematica* manipulation commands on tensorial expressions, we must have unique dummy indices internally when multiplying

expressions and other manipulations, etc. Thus, inexorably, internal uniqueness must be maintained. There are two solutions to this uniqueness problem: (*i*) to furnish tools or methods by which the user can keep track of the dummy indices and (*ii*) to overload the basic operations like `Power` and `Times`. We will adopt (*i*) since overloading the basic *Mathematica* functions appreciably slows down the overall system.

The tools we provide can take several forms. One form is the introduction of a new multiplication that is "tensor friendly". By this stage, introducing the notation for such an object should be somewhat routine. Here is one possible infix notation.

```
In[16]:= InfixNotation[*T, TensorTimes]
```

We can easily provide some simple semantics for this new operation, `TensorTimes`. We change all dummy indices of each argument into unique dummy indices via the function `Dummify` introduced by `Tensors`. Then normal multiplication is performed on the arguments, and finally the resulting expression is reindexed back again.

```
In[17]:= TensorTimes @ args___ := ReIndex[Times @@ Dummify /@ {args}]
```

This is now adequate to solve our uniqueness problems.

```
In[18]:= (T^i T_i) *T (T^i T_i)

Out[18]= T^a T^b T_a T_b
```

Furthermore, it is comparably trivial to introduce a new power operation, say `TensorPower`, which is similarly "tensor friendly".

The above approach is somewhat flawed in that eventually we must use normal multiplication and powers in order to use functions like `Simplify`, `Factor`, etc. Of course, we could write such versions ourselves with the basic inheritance paradigm, but that approach would be somewhat isolationist. It is conceivable that in some languages, possibly AXIOM, or maybe the extension GAUSS[243] for Maple[58, 152, 244], one may be able to define one's own multiplication that seamlessly interoperates with the simplification functions of the underlying language without any loss of efficiency. In this case, it would be ideal to introduce one's own tailored tensor multiplications, powers, factoring, simplifications, etc.

Of course, with the developments and language modifications previously presented in this thesis, it should be evident that we could in fact introduce new structures that perform tensor multiplications which incorporate automatic reindexing. But even this would still not provide *true* "interoperability". Notably though, that we have to trouble ourselves with such exceptions and that it is not possible to fix them in a "nice" way is testament to the fact that there is still a large scope for improvement in the underlying language.

The alternative solution, which we previously superficially presented in §3.5.3 *Dummy Indices*, is greatly preferable. It is much like delayed assignment in *Mathematica*, but we need the dummy index to be dummified in a delayed manner.

```
In[19]:= expr := Module[{i}, T^i T_i]
```

Now multiplying `expr` by `expr` maintains the correct dummy indices.

```
In[20]:= expr expr

Out[20]= T^i$1298 T^i$1299 T_i$1298 T_i$1299
```

One common function that almost all of the tensor packages seem to share is a function to *reindex* an expression. For instance, MathTensor[258], Ricci[202], TTC[15, 56], EinS[192, 193], etc., all seem to have such a function. As we saw in §3.5.3 *Dummy Indices*, the *Tensors* package is no exception. Although our canonicalization algorithm performs reindexing as a by-product of canonicalization, it is still faster to reindex expressions if this is all that is required. In order to reindex an expression, the dummy indices being used must be declared. For instance, the last expression can be reindexed, since any index of the form i$*num* is known to be a general index.

```
In[21]:= ReIndex @ %

Out[21]= T^a T^b T_a T_b
```

Yet, when the index class of an index is not known, then reindexing cannot take place.

```
In[22]:= ReIndex @ T^Δ T_Δ

Out[22]= T^Δ T_Δ
```

Therefore, we have achieved an acceptable level of functionality by using modules. Yet, despite achieving unique dummy indices, explicitly using a module in every definition involving tensors is somewhat ugly. The dummy indices must be specified every time, even though we should *easily* be able to work such things out. As we saw in §3.5.3 *Dummy Indices*, we can incorporate this automatic relabeling into a new assignment. This is the topic of the next subsection.

7.2.3 Tensorial Assignments

It should be clear after the motivation of the previous subsection that we need to create a new "tensorial assignment" operation which automatically incorporates unique dummy indices. Here are the notations for such assignments.

function	*notation*	*explanation*
`TensorSet [lhs, rhs]`	*lhs* ≡ *rhs*	set the *lhs* to be equal to the *rhs* where all summed indices on the *rhs* will be replaced by new unique indices
`TensorSetDelayed [lhs, rhs]`	*lhs* :≡ *rhs*	set the *lhs* to be equal to the delayed value of *rhs*, where all summed indices on the *rhs* will be replaced by new unique indices

`TaggedTensorSet [lhs, rhs, tag]`	*tag* / : *lhs* ≅ *rhs*	set the *lhs* to be equal to the *rhs*, but associate the rule with *tag*. All summed indices on the *rhs* will be replaced by new unique indices
`TaggedTensorSetDelayed [lhs, rhs, tag]`	*tag* / : *lhs* :≅ *rhs*	set the *lhs* to be equal to the delayed value of *rhs*, but associate the rule with *tag*. All summed indices on the *rhs* will be replaced by new unique indices
`DynamicTensorSet [lhs, rhs, tag, env]`	*env* ↗ *tag* / : *lhs* ≅ *rhs*	set the *lhs* to be equal to the *rhs* within the environment *env*, but associate the rule with *tag*. All summed indices on the *rhs* will be replaced by new unique indices
`DynamicTensorSetDelayed [lhs, rhs, tag, env]`	*env* ↗ *tag* / : *lhs* :≅ *rhs*	set the *lhs* to be equal to the delayed value of *rhs* within the environment *env*, but associate the rule with *tag*. All summed indices on the *rhs* will be replaced by new unique indices

Syntax of tensor assignment functions.

Creating such notations as in the table above should be routine by now. (We previously used such assignments for our "associative" dynamic rules in §5 *Prototypical Structures and Quantum Mechanics*, and have seen many other similar examples.) The new "tensor" assignments have the following sorts of notations.

```
In[23]:= Notation[ℓhs_ :≅ rhs_ ⟺ TensorSetDelayed[ℓhs_, rhs_]];
         Notation[ℓhs_ ≅ rhs_ ⟺ TensorSet[ℓhs_, rhs_]];
```

Our "tensor" assignments should work with tags, just like normal *Mathematica* functions.

```
In[25]:= Notation[tag_ / : ℓhs_ :≅ rhs_ ⟺ TaggedTensorSetDelayed[tag_, ℓhs_, rhs_]];
         Notation[tag_ / : ℓhs_ ≅ rhs_ ⟺ TaggedTensorSet[tag_, ℓhs_, rhs_]];
```

Indeed, for proper handling, we need our tensorial assignments to be dynamical. These cases can easily be handled by notations like the following.

```
In[27]:= Notation[env_ ↗ tag_ / : ℓhs_ :≅ rhs_ ⟺
              DynamicTaggedTensorSetDelayed[env_, tag_, ℓhs_, rhs_]];
```

The *Prototypes* package includes the rest of the notations associated with tensorial assignment, and also the notations for the corresponding rules to the above assignments. As in the case of the "associative" structures of §5 *Prototypical Structures and Quantum Mechanics*, these heads are non-persistent. That is, any statements involving such tensorial assignments or rules are immediately transformed to modified assignments or modified rules. The complete list of structures are: `TensorSet`, `TensorSetDelayed`, `TaggedTensorSet`, `TaggedTensorSetDelayed`, `DynamicTensorSet`, `DynamicTensorSetDelayed`, `TensorRule`, `TensorRuleDelayed`, `TaggedTensorRule`, `TaggedTensorRuleDelayed`, `DynamicTensorRule`, `DynamicTensorRuleDelayed`.

These functions can all be fairly simply encoded using our notations and the function DummyIndices. For instance, here is the definition for the DynamicTensorSetDelayed.

```
In[28]:= (e_ ↗ t_ /: lhs_ :≔ rhs_) :=
           With[{dummies = DummyIndices[rhs_u] \ DummyIndices[lhs_u]},
             e_u ↗ t_u /: lhs_u := Module[dummies, rhs_u]]
```

Thus, there should be nothing mysterious about our definitions for tensorial assignments.

For instance, as we presented earlier in §3.5.2 *Simple Examples Using Tensors*, we can expand the Christoffel symbol in terms of the metric.

```
In[29]:= ResolveΓ ↗ Γ /: Γ^λ_ _α_ _β_ :≔ 1/2 g^λ μ (g_α μ, β + g_μ β, α - g_α β, μ)
         ResolveΓ @ any_ := DynamicBar @ any
```

Similarly, the Riemann tensor can be expanded as follows — see [81, 240, 255, 326, 333, etc]. (We previously mentioned this in §6.14.1 *Origins of Linear Symmetries.*)

```
In[31]:= ResolveR ↗ R /: R^λ_ _ρ_ _μ_ _ν_ :≔
           Γ^λ_ρ ν, μ - Γ^λ_ρ μ, ν + Γ^α_ρ ν Γ^λ_α μ - Γ^α_ρ μ Γ^λ_α ν
         ResolveR @ any_ := DynamicBar @ any
```

If we combine both of these environments into a new environment, we can perform resolutions.

```
In[33]:= Assign[Values @ ResolveΓ +{} Values @ ResolveR /. {
                  ResolveΓ → Resolve,
                  ResolveR → Resolve}]
```

Let us now resolve a typical Riemannian tensor.

```
In[34]:= Resolve[R^α_β γ δ]
```

$$\text{Out[34]}= \frac{1}{4} g^{\alpha\,\mu\$1380}\, g^{\alpha\$1378\,\mu\$1379} \left(-g_{\beta\,\delta,\,\mu\$1379} + g_{\beta\,\mu\$1379,\,\delta} + g_{\mu\$1379\,\delta,\,\beta}\right) \left(-g_{\alpha\$1378\,\gamma,\,\mu\$1380} + g_{\alpha\$1378\,\mu\$1380,\,\gamma} + g_{\mu\$1380\,\gamma,\,\alpha\$1378}\right) - \frac{1}{4} g^{\alpha\,\mu\$1382}\, g^{\alpha\$1378\,\mu\$1381} \left(-g_{\beta\,\gamma,\,\mu\$1381} + g_{\beta\,\mu\$1381,\,\gamma} + g_{\mu\$1381\,\gamma,\,\beta}\right) \left(-g_{\alpha\$1378\,\delta,\,\mu\$1382} + g_{\alpha\$1378\,\mu\$1382,\,\delta} + g_{\mu\$1382\,\delta,\,\alpha\$1378}\right) - \Gamma^{\alpha}_{\beta\,\gamma,\,\delta} + \Gamma^{\alpha}_{\beta\,\delta,\,\gamma}$$

```
In[35]:= ReIndex @ Expand @ %
```

```
Out[35]= 1/4 g^(α η) g^(ϵ ζ) g_(β δ, ζ) g_(ϵ γ, η) - 1/4 g^(α η) g^(ϵ ζ) g_(β ζ, δ) g_(ϵ γ, η) -
  1/4 g^(α η) g^(ϵ ζ) g_(β γ, ζ) g_(ϵ δ, η) + 1/4 g^(α η) g^(ϵ ζ) g_(β ζ, γ) g_(ϵ δ, η) -
  1/4 g^(α η) g^(ϵ ζ) g_(β δ, ζ) g_(ϵ η, γ) + 1/4 g^(α η) g^(ϵ ζ) g_(β ζ, δ) g_(ϵ η, γ) +
  1/4 g^(α η) g^(ϵ ζ) g_(β γ, ζ) g_(ϵ η, δ) - 1/4 g^(α η) g^(ϵ ζ) g_(β ζ, γ) g_(ϵ η, δ) +
  1/4 g^(α η) g^(ϵ ζ) g_(ϵ δ, η) g_(ζ γ, β) - 1/4 g^(α η) g^(ϵ ζ) g_(ϵ η, δ) g_(ζ γ, β) -
  1/4 g^(α η) g^(ϵ ζ) g_(ϵ γ, η) g_(ζ δ, β) + 1/4 g^(α η) g^(ϵ ζ) g_(ϵ η, γ) g_(ζ δ, β) -
  1/4 g^(α η) g^(ϵ ζ) g_(β δ, ζ) g_(η γ, ϵ) + 1/4 g^(α η) g^(ϵ ζ) g_(β ζ, δ) g_(η γ, ϵ) +
  1/4 g^(α η) g^(ϵ ζ) g_(ζ δ, β) g_(η γ, ϵ) + 1/4 g^(α η) g^(ϵ ζ) g_(β γ, ζ) g_(η δ, ϵ) -
  1/4 g^(α η) g^(ϵ ζ) g_(β ζ, γ) g_(η δ, ϵ) - 1/4 g^(α η) g^(ϵ ζ) g_(ζ γ, β) g_(η δ, ϵ) - Γ^α_(β γ, δ) + Γ^α_(β δ,
```

Therefore, the expansion occurs as one would hope. Yet there is a problem: the partial derivatives of the Christoffel symbols have not been resolved. Shortly we will see how to easily rectify this matter; but first, let us introduce the remaining manipulation functions of the *Tensors* package.

7.2.4 Manipulation Functions of the Tensors Package

Let us formally present the functions that the *Tensors* package provides. These are summarized as follows.

`UsedIndices[`*expr*`]`	returns the indices used in the expression *expr*.
`ReIndex[`*expr*`]`	reindex all dummy indices in the expression *expr* to canonical indices
`ExpandContraction[`*expr*`]`	expand out any contracted dummy indices into a sum of terms where the dummy indices range over their allowed coordinates
`Dummify[`*expr*`]`	reindex all dummy indices in the expression *expr* into unique dummy indices which have not been used previously
`DummyIndices[`*expr*`]`	returns the list of dummy indices occurring in the expression *expr*

The index handling functions.

The function `DummyIndices` will give all dummy indices in an expression. For instance,

```
In[36]:= DummyIndices[T^a T^b T_a T_b + f[g_mn T^m T^n]]
```

```
Out[36]= {a, b, m, n}
```

Yet, when the head of an expression is not recognized as a "multiplicative head", then the dummy indices are not "recognized".

```
In[37]:= DummyIndices[T^a T^b T_a T_b + f[g_mn, T^m T^n]]
```

```
Out[37]= {a, b}
```

In contrast, `UsedIndices` recognize all indices at any depth.

```
In[38]:= UsedIndices[T^a T^b T_a T_b + f[g_mn, T^m T^n]]
```

```
Out[38]= {m, n, m, n, a, b, a, b}
```

The fact that `DummyIndices` does not recognize indices at every depth makes intuitive sense, since one would not expect say `DummyIndices[Hold[...]]` to return any dummy indices. Thus, we need to be able to determine when a head should be treated as a tensor multiplicative head. The following function is provided for this case.

`DeclareTensorialMultiplicative[` *head*`]`	declares that the head *head* is to be treated as a multiplicative head to be canonicalized over
`DeclareTensorialMultiplicative[` *head*`, Commutative → True | False]`	declares that the head *head* is to be treated as a multiplicative head to be canonicalized over, and treated as commutative according to the value of the option

The function for declaring that a head should be treated as multiplicative by the *Tensors* package.

If we now declare `f` to be a tensorial multiplicative head, `DummyIndices` will consider indices separated within the top level of an expression with head f to be dummy indices. Let us demonstrate this.

```
In[39]:= DeclareTensorialMultiplicativeHead[f, Commutative → True]
```

```
In[40]:= DummyIndices[T^a T^b T_a T_b + f[g_mn, T^m T^n]]
```

```
Out[40]= {a, b, m, n}
```

Moreover, `Canonicalize` will treat the head `f` in the same way. For instance,

```
In[41]:= Canonicalize[f[T^a, T^b, g_ab] + f[g_mn, T^m, T^n]]
```

```
Out[41]= 2 f[g^ab, T_a, T_b]
```

In fact, we saw this exact same behavior previously for our non-commutative times in §3.5.5 *Simple Cartesian Vector Calculus I.*

7.2.5 Partial and Covariant Derivatives

Let us return to the issue of the handling of derivatives. Due to our canonicalization algorithm and the tensor parsing and formatting, it is easier to make the partial derivatives as well as the covariant derivatives part of the structure of the tensor itself, that is, something like $T^{\alpha\ldots}{}_{\beta\ldots,\mu}$. Henceforth, let us call this form of a derivative an *embedded form*, since the derivative, like the other indices, is part of the tensor structure itself. This was discussed to some extent in §3.4.3 *Prototypical Tensor Expression Structure*.

As we saw previously, the Christoffel symbols with partial derivatives were not being handled by our rule for resolving Christoffel symbols. To reiterate, without partial derivatives, resolution works as intended.

```
In[42]:= Resolve[Γ^α_{βγ}]
```

$$\text{Out[42]}= \frac{1}{2}\, g^{\alpha\,\mu\$1392}\left(-g_{\beta\gamma,\,\mu\$1392}+g_{\beta\,\mu\$1392,\,\gamma}+g_{\mu\$1392\,\gamma,\,\beta}\right)$$

Yet as we saw, when partial derivatives are included in the tensor, resolution does not work.

```
In[43]:= Resolve[Γ^α_{βγ,δ}]
```

$$\text{Out[43]}= \Gamma^{\alpha}{}_{\beta\gamma,\,\delta}$$

This, of course, has to do with the full form of the tensor expression. What is needed is a general *"explicit"* partial derivative. First, we introduce the notation for explicit tensor partial derivatives.

```
In[44]:= Notation[∂̄_p_ expr_ ⟺ TensorD[expr_, p_]]
         Notation[∂̄^p_ expr_ ⟺ TensorD[expr_, p_^+]]
         Notation[∂̄_p_ expr_ ⟺ TensorD[expr_, p_]]
```

We now inherit from the prototype of a generic derivative to create our tensor partial derivative.

```
In[47]:= Assign [ GenericD_Rules /. {
                    GenericD → TensorD,
                    GenericTimes → Times,
                    GenericPlus → Plus,
                    GenericConstQ → ConstQ,
                    Generic_Env → Act} ]
         Act @ any_ := DynamicBar @ any
```

To illustrate the action of our new tensorial derivative, consider the following example.

```
In[49]:= ∂̄^μ (∂̄_μ (a^2 - b c))
```

$$\text{Out[49]}= \bar{\partial}^{\mu}(\bar{\partial}_{\mu}\, a^2) - \bar{\partial}^{\mu}(\bar{\partial}_{\mu}\,(b\,c))$$

We see that by the design of the generic rules, the linearity of the derivative is always active. This, of course, can be changed if desired, so that the "activity" of the linearity rules is restricted to a specific environment. It is also clear that the "product rule" is not always active. Indeed, by design, the distribution of derivatives over powers and products is restricted to being active in the `Act` environment.

```
In[50]:= Act @ %
```

$$\text{Out[50]= } -\bar{\partial}_\mu b\, \bar{\partial}^\mu c - \bar{\partial}^\mu b\, \bar{\partial}_\mu c + 2\,(\bar{\partial}^\mu a\, \bar{\partial}_\mu a + a\, \bar{\partial}^\mu (\bar{\partial}_\mu a)) - c\, \bar{\partial}^\mu (\bar{\partial}_\mu b) - b\, \bar{\partial}^\mu (\bar{\partial}_\mu c)$$

Actually, it is again convenient to have `Act` be an "expanding" environment. This is accomplished by simply inheriting the appropriate rules of `Expand`.

```
In[51]:= Assign[Select[Values @ Expand, (NonCommutativeTimes) ∈?∞ rule &_rule] /.
           Expand → Act]
```

We can also make our tensorial derivative active over non-commutative times objects.

```
In[52]:= Assign [ GenericD_Rules /. {
                    GenericD → TensorD,
                    GenericTimes → NonCommutativeTimes,
                    GenericPlus → Plus,
                    GenericConstQ → ConstQ,
                    Generic_Env → Act} ]
```

```
In[53]:= ∂̿_μ (a^2 · b)
```

$$\text{Out[53]= } \bar{\bar{\partial}}_\mu (a^2 \cdot b)$$

```
In[54]:= Act @ %
```

$$\text{Out[54]= } a \cdot a \cdot \bar{\bar{\partial}}_\mu b + a \cdot \bar{\partial}_\mu a \cdot b + \bar{\bar{\partial}}_\mu a \cdot a \cdot b$$

```
In[55]:= Simplify @ %
```

$$\text{Out[55]= } a^2 \cdot \bar{\bar{\partial}}_\mu b + a \cdot \bar{\partial}_\mu a \cdot b + \bar{\bar{\partial}}_\mu a \cdot a \cdot b$$

Yet, since our $\bar{\partial}_\mu$ now works over both non-commutative and commutative objects, we obtain some slightly unwanted behavior.

```
In[56]:= Act[∂̿_μ a^2]
```

$$\text{Out[56]= } a \cdot \bar{\bar{\partial}}_\mu a + \bar{\bar{\partial}}_\mu a \cdot a$$

Of course, if we declared the various objects in our expressions as semi-commutative multipliers, then everything would work as desired. However, it is only because the rule for `NonCommutativeTimes` power expansions was declared last and replaces the similar earlier rule for `Times`, that we obtain this behavior. We could easily change our prototypical rule set for non-commutative powers; but for simplicity, let us just reinherit the rules for differentation involving `Times`.

```
In[57]:= Assign [ GenericD_Rules /. {
                    GenericD → TensorD,
                    GenericTimes → Times,
```

```
                GenericPlus → Plus,
                GenericConstQ → ConstQ,
                Generic$_{Env}$ → Act} ]
```

```
In[58]:= ∂̄_μ (a^2 · b) // Act
Out[58]= a · a · ∂̄_μ b + a · ∂̄_μ a · b + ∂̄_μ a · a · b

In[59]:= ∂̄_μ a^2 // Act
Out[59]= 2 a ∂̄_μ a
```

Of course, there are many variations on the above behavior. For instance, instead of expansion-like behavior for `Act`, we might instead prefer to change the generic prototypes for non-commutative differentation. In this case, we would just change the rules for *Generic*`DPower`$_{\texttt{Rules}}$. In any case, with some simple changes, the desired behavior follows. This illustrates the supreme flexibility of our approach. Let us observe the rules for derivatives working in the following.

```
In[60]:= ∂̄^μ (∂̄_μ (a^2 - b c))
Out[60]= ∂̄^μ (∂̄_μ a^2) - ∂̄^μ (∂̄_μ (b c))

In[61]:= FullForm @ %
Out[61]//FullForm=
Plus[TensorD[TensorD[Power[a, 2], Low[μ]], High[μ]],
   Times[-1, TensorD[TensorD[Times[b, c], Low[μ]], High[μ]]]]

In[62]:= Expand @ Act @ %
Out[62]= 2 ∂̄^μ a ∂̄_μ a - ∂̄_μ b ∂̄^μ c - ∂̄^μ b ∂̄_μ c + 2 a ∂̄^μ (∂̄_μ a) - c ∂̄^μ (∂̄_μ b) - b ∂̄^μ (∂̄_μ c)
```

Before progressing on, let us add "explicit" covariant derivatives to our system. The covariant derivative is defined for any tensor $T^{\alpha\ldots}{}_{\beta\ldots}$ as follows — see [81, 255, 326, 333] for further background.

$$\bar{\nabla}_\mu T^{\alpha\ldots}{}_{\beta\ldots} = T^{\alpha\ldots}{}_{\beta\ldots,\mu} + \Gamma^{\alpha}{}_{\mu\nu} T^{\nu\ldots}{}_{\beta\ldots} + \ldots - \Gamma^{\nu}{}_{\mu\beta} T^{\alpha\ldots}{}_{\nu\ldots} - \ldots \qquad (7.2.a)$$

We introduce covariant derivatives in almost exactly the same way as was done for the tensorial partial derivative. First the notation.

```
In[63]:= Notation[∇̄_ρ_ expr_ ⟺ TensorCovD[expr_, ρ_^-]]
         Notation[∇̄^ρ_ expr_ ⟺ TensorCovD[expr_, ρ_^+]]
         Notation[∇̄_ρ_ expr_ ⟺ TensorCovD[expr_, ρ_]]
```

The covariant derivative is also a "differentation" operator, and so again we inherit from the prototype of generic differentiation.

```
In[66]:= Assign [ GenericD$_{Rules}$ /. {
                GenericD → TensorCovD,
                GenericTimes → Times,
                GenericPlus → Plus,
                GenericConstQ → ConstQ,
                Generic$_{Env}$ → Act} ]
```

```
In[67]:= ∇̄^μ (∇̄_μ (a - b c - 1))

Out[67]= ∇̄^μ (∇̄_μ a) - ∇̄^μ (∇̄_μ (b c))

In[68]:= Act @ %

Out[68]= -∇̄_μ b ∇̄^μ c - ∇̄^μ b ∇̄_μ c + ∇̄^μ (∇̄_μ a) - c ∇̄^μ (∇̄_μ b) - b ∇̄^μ (∇̄_μ c)
```

By examining the above, one realizes that some of these terms should be able to be canonicalized to the same term. In fact, the same statement was also true of the analogous partial derivative expression above. Therefore, further to all of the other rules above, we should declare that `TensorD` and `TensorCovD` are both tensor multiplicative heads, so that it is permissible to reindex over these derivative operations.

```
In[69]:= DeclareTensorialMultiplicativeHead[TensorD, Commutative → False]
         DeclareTensorialMultiplicativeHead[TensorCovD, Commutative → False]

In[71]:= ReIndex [%%%]

Out[71]= -∇̄_α b ∇̄^α c - ∇̄^α b ∇̄_α c + ∇̄^α (∇̄_α a) - c ∇̄^α (∇̄_α b) - b ∇̄^α (∇̄_α c)
```

Clearly, the variables of differentiation have been "reindexed" as desired. In the next subsection we introduce mechanisms to transform back and forth from the explicit form of the derivatives to the embedded form of the derivatives.

On a final note for this section, it should be evident from the above that it would be extremely simple to create an "explicit" Lie derivative, in addition to the explicit partial and covariant derivatives.

7.2.6 Embedded versus Explicit Derivatives

In the previous subsection, we presented the "explicit" forms for the partial and covariant derivatives. Let us now introduce two generic prototypes to transform from the "embedded" derivatives to the "explicit" derivatives.

```
In[72]:= explicitizePartialDerivatives_Rules := {
           Generic_Env ↗ (T_(i___), p_) :↦ ∂̄_p T_(i), ,
           Generic_Env ↗ (T_(i___), par___, p_) :↦ ∂̄_p T_(i, par) }

In[73]:= implicitizePartialDerivatives_Rules := {
           Generic_Env ↗ (∂̄_p_ T_(i___), par___) :↦ T_(i, par, p), ,
           Generic_Env ↗ (∂̄_p_ T_(i___)) :↦ T_(i, p) }
```

Let us instantiate these prototypes to two simple transformation functions.

```
In[74]:= Assign[implicitizePartialDerivatives_Rules /. Generic_Env → implicitizePD]
         Assign[explicitizePartialDerivatives_Rules /. Generic_Env → explicitizePD]
         implicitizePD @ any_ := DynamicBar @ ReIndex @ any
         explicitizePD @ any_ := DynamicBar @ ReIndex @ any
```

We can now easily solve the problem raised back in §7.2.3 *Tensorial Assignments*. Recall that there we resolved the Riemann tensor into Christoffel symbols, which in turn were resolved into metric tensors and derivatives thereof. If we now make our resolving environment "explicitize" partial derivatives, then the normal formulas for resolving will once again be directly applicable. It is also convenient to have the resolutions be "expanding". Moreover, it is convenient to create a generic set of resolution rules, so that we may use them later in specific environments.

```
In[78]:= GenericResolve_Rules =
           (explicitizePartialDerivatives_Rules /. Generic_Env → GenericResolve) +{}
             (Select[Values @ Expand, (NonCommutativeTimes) ∈?∞ rule &_rule] /.
                   Expand → GenericResolve) +{}
             (GenericD_Rules /. {
                        GenericD → TensorD,
                        GenericTimes → Times,
                        GenericPlus → Plus,
                        GenericConstQ → ConstQ,
                        Generic_Env → GenericResolve});
```

Let us next inherit these rules to `Resolve`.

```
In[79]:= Assign[GenericResolve_Rules /. GenericResolve → Resolve]
```

Resolutions are now carried through to completion.

```
In[80]:= implicitizePD @ Resolve @ R^α_βγδ
```

$$\text{Out[80]}= \frac{1}{4} g^{\alpha\eta} g^{\epsilon\zeta} \left(-g_{\beta\delta,\zeta} + g_{\beta\zeta,\delta} + g_{\zeta\delta,\beta}\right)\left(-g_{\epsilon\gamma,\eta} + g_{\epsilon\eta,\gamma} + g_{\eta\gamma,\epsilon}\right) - \frac{1}{4} g^{\alpha\eta} g^{\epsilon\zeta} \left(-g_{\beta\gamma,\zeta} + g_{\beta\zeta,\gamma} + g_{\zeta\gamma,\beta}\right)\left(-g_{\epsilon\delta,\eta} + g_{\epsilon\eta,\delta} + g_{\eta\delta,\epsilon}\right) + \frac{1}{2}\left(-g^{\alpha\epsilon}{}_{,\delta}\left(-g_{\beta\gamma,\epsilon} + g_{\beta\epsilon,\gamma} + g_{\epsilon\gamma,\beta}\right) - g^{\alpha\epsilon}\left(-g_{\beta\gamma,\epsilon\delta} + g_{\beta\epsilon,\gamma\delta} + g_{\epsilon\gamma,\beta\delta}\right)\right) + \frac{1}{2}\left(g^{\alpha\epsilon}{}_{,\gamma}\left(-g_{\beta\delta,\epsilon} + g_{\beta\epsilon,\delta} + g_{\epsilon\delta,\beta}\right) + g^{\alpha\epsilon}\left(-g_{\beta\delta,\epsilon\gamma} + g_{\beta\epsilon,\delta\gamma} + g_{\epsilon\delta,\beta\gamma}\right)\right)$$

Similarly, we can perform the same sorts of transformations for covariant derivatives.

```
In[81]:= explicitizeCovariantDerivatives_Rules := {
           Generic_Env ↗ (T_i___ ; p_) :↦ ∇̄_p T_i ,
           Generic_Env ↗ (T_i___ ; par___, p_) :↦ ∇̄_p T_i ; par /; ", " ∉? {par}}
```

```
In[82]:= implicitizeCovariantDerivatives_Rules := {
  Generic_Env ↗ (∇̄_p_ T_i___ ; par___) :> T_i ; par p /; "," ∉ {par},
  Generic_Env ↗ (∇̄_p_ T_i___) :> T_i ; p /; "," ∉ {i}}
```

Finally, let us also create an environment for expanding covariant derivatives into partial derivatives according to (7.2.a). It is convenient to perform the expansions in the embedded derivative form. The following code is somewhat contorted, since targeted replacements need to be made throughout copies of the structure.

```
In[83]:= expandCovariantDerivatives_Rules := {
  Generic_Env ↗ (T_i___ ;) :> T_i ,
  Generic_Env ↗ (T_i___ ; c___^λ_) :> g^(λ μ) T_i ; c μ ,
  Generic_Env ↗ (T_i___ λ_ /; ";"^- ∈ {i} && "," ∉ {i}) :> ∂̄_λ T_i +
      Module[{μ}, Plus @@ Table[With[{α = {i}[[j]]},
          Γmul[α, μ, λ] MapAt[replaceIndex[α, μ] &,
            (T_i)_u, {2, j}]], {j, 1, {i}_len}]]};
```

This code relies on the following routines.

```
In[84]:= Γmul[";"^-, μ_, λ_] := 0
  Γmul[α_^-, μ_, λ_] := -Γ^μ_(α λ)
  Γmul[α_^+, μ_, λ_] := +Γ^α_(μ λ)
  replaceIndex[α_^+, μ_] := μ^+
  replaceIndex[α_^-, μ_] := μ^-
```

Let us now create a simple function for transforming all covariant derivatives in a tensorial expression to partial derivatives.

```
In[89]:= Assign[explicitizePartialDerivatives_Rules +{}
            expandCovariantDerivatives_Rules /. Generic_Env → covariantsToPartia
In[90]:= Assign[EnvironmentValues @ Act /. Act → covariantsToPartials]
In[91]:= covariantsToPartials @ any_ := DynamicBar @ ReIndex @ Expand @ any
```

Here is a simple example of our expansion.

```
In[92]:= covariantsToPartials @ T_β ; μ
Out[92]= -T_α Γ^α_(β μ) + ∂̄_μ T_β
```

For yet another example of why it is extremely useful to use generic prototypes, consider the following. Since `Canonicalize` works on the embedded form of the partial and covariant

derivatives, it is natural to transform any explicit derivatives into an embedded form under the `Canonicalize` environment. This is simply accomplished by inheriting from the generic implicitizing prototypes.

```
In[93]:= Assign[implicitizePartialDerivatives_Rules +{}
                  implicitizeCovariantDerivatives_Rules /.
           Generic_Env → Canonicalize]
```

As a preliminary to our next example, let us declare that the Christoffel symbols are symmetric on their second and third indices, since we are working with a metric connection [81, 326, 333].

```
In[94]:= DeclareSymmetries[Γ, 3, {(2 ↔ 3)+}]
```

Here is an example calculation involving the "symmetric difference" of a covariant derivative.

```
In[95]:= covariantsToPartials [T_{β;μ} - T_{μ;β}]
```

$$\text{Out[95]= } -\mathbf{T}_{\alpha}\,\Gamma^{\alpha}{}_{\beta\mu} + \mathbf{T}_{\alpha}\,\Gamma^{\alpha}{}_{\mu\beta} + \bar{\partial}_{\mu}\mathbf{T}_{\beta} - \bar{\partial}_{\beta}\mathbf{T}_{\mu}$$

By canonicalizing the above expression, the explicit forms of the partial derivatives are transformed into the embedded partial derivatives.

```
In[96]:= Canonicalize @ %
```

$$\text{Out[96]= } \mathbf{T}_{\beta,\mu} - \mathbf{T}_{\mu,\beta}$$

Thus, we see that the "differences" of covariant derivatives are equivalent to the "differences" of partial derivatives.

7.2.7 Partial Derivative Handling

In this subsection we perform some simple calculations involving the derivative structures we have built up throughout the past two subsections. Let us first show that commuting two covariant derivatives of a covariant vector leads to a term involving the Riemann tensor.

```
In[97]:= covariantsToPartials [T_{β;μν} - T_{β;νμ}]
```

$$\begin{aligned}\text{Out[97]= } &\mathbf{T}_{\gamma}\,\Gamma^{\alpha}{}_{\beta\nu}\,\Gamma^{\gamma}{}_{\alpha\mu} - \mathbf{T}_{\gamma}\,\Gamma^{\alpha}{}_{\beta\mu}\,\Gamma^{\gamma}{}_{\alpha\nu} + \mathbf{T}_{\gamma}\,\Gamma^{\alpha}{}_{\mu\nu}\,\Gamma^{\gamma}{}_{\beta\alpha} - \mathbf{T}_{\gamma}\,\Gamma^{\alpha}{}_{\nu\mu}\,\Gamma^{\gamma}{}_{\beta\alpha} - \\ &\Gamma^{\alpha}{}_{\mu\nu}\,\bar{\partial}_{\alpha}\mathbf{T}_{\beta} + \Gamma^{\alpha}{}_{\nu\mu}\,\bar{\partial}_{\alpha}\mathbf{T}_{\beta} - \mathbf{T}_{\alpha}\,\bar{\partial}_{\nu}\Gamma^{\alpha}{}_{\beta\mu} + \mathbf{T}_{\alpha}\,\bar{\partial}_{\mu}\Gamma^{\alpha}{}_{\beta\nu} + \bar{\partial}_{\nu}\left(\bar{\partial}_{\mu}\mathbf{T}_{\beta}\right) - \bar{\partial}_{\mu}\left(\bar{\partial}_{\nu}\mathbf{T}_{\beta}\right.\end{aligned}$$

```
In[98]:= R^α_{βμν} T_α // ResolveR // ReIndex
```

$$\text{Out[98]= } \mathbf{T}_{\alpha}\left(-\Gamma^{\alpha}{}_{\gamma\nu}\,\Gamma^{\gamma}{}_{\beta\mu} + \Gamma^{\alpha}{}_{\gamma\mu}\,\Gamma^{\gamma}{}_{\beta\nu} - \Gamma^{\alpha}{}_{\beta\mu,\nu} + \Gamma^{\alpha}{}_{\beta\nu,\mu}\right)$$

```
In[99]:= % == %% // Canonicalize
```

```
Out[99]= True
```

A similar relation is also true of the commutation of the covariant derivatives of a contravariant tensor.

```
In[100]:= covariantsToPartials[ T^β_{;μν} - T^β_{;νμ} ] == -R^β_{αμν} T^α // ResolveR //
            Canonicalize
Out[100]= True
```

In fact, the commutation of covariant derivatives leads to a Riemann tensor for each index in the base tensor.

```
In[101]:= covariantsToPartials[ T^α_{βγδ;μν} - T^α_{βγδ;νμ} ] ==
            R^τ_{βμν} T^α_{τγδ} + R^τ_{γμν} T^α_{βτδ} + R^τ_{δμν} T^α_{βγτ} -
              R^α_{τμν} T^τ_{βγδ} // ResolveR // Canonicalize
Out[101]= True
```

These sorts of equations can be used to generate auxiliary equations for our tensor canonicalization algorithm.

Let us consider another example that we encountered earlier.

```
In[102]:= ∂̄^μ (∂̄_μ (a^2 - b c))
Out[102]= ∂̄^μ (∂̄_μ a^2) - ∂̄^μ (∂̄_μ (b c))
```

If we make the partial derivatives act on their arguments, we obtain the following expansion.

```
In[103]:= Expand @ Act @ %
Out[103]= 2 ∂̄^μ a ∂̄_μ a - ∂̄_μ b ∂̄^μ c - ∂̄^μ b ∂̄_μ c + 2 a ∂̄^μ (∂̄_μ a) - c ∂̄^μ (∂̄_μ b) - b ∂̄^μ (∂̄_μ c)
```

Some of the terms in the above expression can be combined.

```
In[104]:= Canonicalize @ %
Out[104]= 2 a_,^α a_,α + 2 a a_,^α_α - b_,^α_α c - 2 b_,^α c_,α - b c_,^α_α
```

```
In[105]:= explicitizePD @ %
Out[105]= 2 ∂̄^α a ∂̄_α a - 2 ∂̄^α b ∂̄_α c + 2 a ∂̄_α (∂̄^α a) - c ∂̄_α (∂̄^α b) - b ∂̄_α (∂̄^α c)
```

It is evident that two terms have combined into the single term $\bar{\partial}^{\alpha}\mathbf{b}\,\bar{\partial}_{\alpha}\mathbf{c}$. Even with the fairly general rules and definitions we have given so far, there are many design choices we need to make at each stage. For instance, when we consider say $\bar{\partial}^{\alpha}(\bar{\partial}_{\alpha}\mathbf{c})$, should we implicitize this to $\mathbf{c}_{,\alpha}^{\ \ \alpha}$? That is, should we make c a tensor? This could easily be accomplished in a single line of code. Yet for now, our design decision will be to not make this change.

Let us repeat the calculation in §6.14.1 *Origins of Linear Symmetries*, that showed that $\mathbf{F}_{\mu\nu,\alpha} + \mathbf{F}_{\nu\alpha,\mu} + \mathbf{F}_{\alpha\mu,\nu} = 0$ for the Maxwell field tensor. This time, instead of defining a direct reduction rule for the Maxwell field, we state how the basic field tensor resolves under a new resolution environment.

```
In[106]:= ResolveF ↗ F /: (F_μ_ν_) := 𝒜_μ;ν - 𝒜_ν;μ
          ResolveF @ any_ := DynamicBar @ any
```

We would like `ResolveF` to be a "resolution environment", so we inherit from the generic prototype *Generic*`Resolve`.

```
In[108]:= Assign[GenericResolve_Rules /. GenericResolve → ResolveF]
```

The fields in the sum can now be resolved as follows.

```
In[109]:= ResolveF[F_μν,α + F_να,μ + F_αμ,ν]
```

$$\text{Out[109]}= \bar{\partial}_\nu \mathcal{A}_{\alpha;\mu} - \bar{\partial}_\mu \mathcal{A}_{\alpha;\nu} - \bar{\partial}_\nu \mathcal{A}_{\mu;\alpha} + \bar{\partial}_\alpha \mathcal{A}_{\mu;\nu} + \bar{\partial}_\mu \mathcal{A}_{\nu;\alpha} - \bar{\partial}_\alpha \mathcal{A}_{\nu;\mu}$$

By changing the covariant derivatives to partial derivatives and canonicalizing, we obtain the desired answer.

```
In[110]:= covariantsToPartials @ %
```

$$\begin{aligned}\text{Out[110]}= {} & -\Gamma^{\beta}_{\mu\nu}\bar{\partial}_\alpha \mathcal{A}_\beta + \Gamma^{\beta}_{\nu\mu}\bar{\partial}_\alpha \mathcal{A}_\beta + \Gamma^{\beta}_{\alpha\nu}\bar{\partial}_\mu \mathcal{A}_\beta - \Gamma^{\beta}_{\nu\alpha}\bar{\partial}_\mu \mathcal{A}_\beta - \\ & \Gamma^{\beta}_{\alpha\mu}\bar{\partial}_\nu \mathcal{A}_\beta + \Gamma^{\beta}_{\mu\alpha}\bar{\partial}_\nu \mathcal{A}_\beta - \mathcal{A}_\beta \bar{\partial}_\nu \Gamma^{\beta}_{\alpha\mu} + \mathcal{A}_\beta \bar{\partial}_\mu \Gamma^{\beta}_{\alpha\nu} + \\ & \mathcal{A}_\beta \bar{\partial}_\nu \Gamma^{\beta}_{\mu\alpha} - \mathcal{A}_\beta \bar{\partial}_\alpha \Gamma^{\beta}_{\mu\nu} - \mathcal{A}_\beta \bar{\partial}_\mu \Gamma^{\beta}_{\nu\alpha} + \mathcal{A}_\beta \bar{\partial}_\alpha \Gamma^{\beta}_{\nu\mu} + \bar{\partial}_\nu\left(\bar{\partial}_\mu \mathcal{A}_\alpha\right) - \\ & \bar{\partial}_\mu\left(\bar{\partial}_\nu \mathcal{A}_\alpha\right) - \bar{\partial}_\nu\left(\bar{\partial}_\alpha \mathcal{A}_\mu\right) + \bar{\partial}_\alpha\left(\bar{\partial}_\nu \mathcal{A}_\mu\right) + \bar{\partial}_\mu\left(\bar{\partial}_\alpha \mathcal{A}_\nu\right) - \bar{\partial}_\alpha\left(\bar{\partial}_\mu \mathcal{A}_\nu\right)\end{aligned}$$

```
In[111]:= Canonicalize @ %
```

$$\text{Out[111]}= 0$$

Thus, as before, we have shown the linear symmetry to be true. In comparison to the previous calculation in §6.14.1 *Origins of Linear Symmetries*, the resolution of the field tensor is specialized to a specific environment, and also the resolution is more general since all partial derivatives are explicitized. The new methods are much cleaner.

7.3 Calculations in General Relativity

In this section we perform some selected calculations which occur in general relativity. We use the calculus tools we built up in the previous section to facilitate this. As will be seen, our calculations are extremely elegant to formulate. Moreover, they highlight both the inheritance paradigm we have developed as well as our canonicalization algorithm. We have so far given a thorough description of our framework. Unfortunately, due to space restrictions, in this applications section we must increase the speed of delivery.

7.3.1 Standard Tensors in General Relativity

In this subsection we collect and enter some standard tensors which are used in general relativity, along with the standard behaviors for these tensors. We of course enter these behaviors using the notations we developed in §7.2.3 *Tensorial Assignments*.

First, the Riemann tensor. Under resolution, we would like all Riemann tensors to be transformed into a default form akin to $\mathbf{R}^{\alpha}{}_{\beta\gamma\delta}$. This can simply be accomplished as follows.

```
In[1]:= ResolveR ↗ R /: (R_λ_ α_ β_ γ_)_hp :≈ g_λ μ R^μ_α β γ

        ResolveR ↗ R /: (R_α_ λ_ β_ γ_)_hp :≈ g^λ μ R_α μ β γ

        ResolveR ↗ R /: (R_α_ β_ λ_ γ_)_hp :≈ g^λ μ R_α β μ γ

        ResolveR ↗ R /: (R_α_ β_ γ_ λ_)_hp :≈ g^λ μ R_α β γ μ
```

Also the Riemann tensor is resolved, as we have previously stated, as follows.

```
In[5]:= ResolveR ↗ R /: (R^λ_ ρ_ μ_ ν_)_hp :≈
          Γ^λ_ρ ν , μ - Γ^λ_ρ μ , ν + Γ^α_ρ ν Γ^λ_α μ - Γ^α_ρ μ Γ^λ_α ν

        ResolveR ↗ R /: (R_ρ_ ν_)_hp :≈ Γ^λ_ρ ν , λ - Γ^λ_ρ λ , ν + Γ^α_ρ ν Γ^λ_α λ - Γ^α_ρ μ Γ^λ_α λ

        ResolveR @ any_ := ReIndex @ Expand @ any
```

Next, the contracted Riemann tensor simplifies to the Ricci tensor.

```
In[8]:= Simplify ↗ R /: (R^β_ _α_ _β_ _γ_)_up := R_α γ

        Simplify ↗ R /: (R^β_ _α_ _γ_ _β_)_up := -R_α γ

        Simplify ↗ R /: (R_α_ ^β_ _β_ _γ_)_up := -R_α γ

        Simplify ↗ R /: (R_α_ ^β_ _γ_ _β_)_up := R_α γ
```

Finally, the contracted Ricci tensor simplifies to the *curvature scalar*.

```
In[12]:= Simplify ↗ R /: (R^α_ _α_)_up := R

         Simplify ↗ R /: (R_α_ ^α_)_up := R
```

Similar to these properties, we can also transform the Christoffel symbols to a normal form under resolution.

```
In[14]:= ResolveΓ ↗ Γ /: (Γ_λ_ _α_ _β_)_up :≈ g_λ μ Γ^μ _α_ _β_

         ResolveΓ ↗ Γ /: (Γ_λ_ ^α_ _β_)_up :≈ g^α μ Γ_λ_ μ _β_

         ResolveΓ ↗ Γ /: (Γ_λ_ _α_ ^β_)_up :≈ g^β μ Γ_λ_ _α_ μ
```

As previously stated, the Christoffel symbols can be resolved into metrics, as follows (Γ is a metric connection [81, 326, 333]).

```
In[17]:= ResolveΓ ↗ Γ /: (Γ^λ_ _α_ _β_)_up :≈ 1/2 g^λ μ (g_α μ , β + g_μ β , α - g_α β , μ)

         ResolveΓ @ any_ := ReIndex @ Expand @ any
```

We want `ResolveΓ` to act like a resolution environment, so we make it inherit from *Generic*`Resolve`. Also, as noted before, the Christoffel symbols are symmetric on their second and third indices, since we are working with a metric connection [81, 326, 333].

```
In[19]:= Assign[GenericResolve_Rules /. GenericResolve → ResolveΓ]
         DeclareSymmetries[Γ, {3, _, _}, {(2 ↔ 3)+}]
```

As mentioned previously, we will use δ both as an index and also to refer to the Kronecker delta. This is a somewhat dangerous thing to do, since we must consequently unprotect it so that we can use it in upvalue assignments.

```
In[21]:= Unprotect[δ];
```

The metric tensor, as well as the Kronecker delta, are symmetric.

```
In[22]:= DeclareSymmetries[g, {2, _, _}, {(1 ↔ 2)+}]
         DeclareSymmetries[δ, {2, _, _}, {(1 ↔ 2)+}]
```

The mixed form of the metric tensor is the Kronecker delta.

```
In[24]:= g /: g^α_ _β_ := δ^α _β
         g /: g _β_^α_ := δ^α _β
         δ /: δ _β_^α_ := δ^α _β
```

The contracted Kronecker delta is equal to the dimensionality of our space, namely d. Also

```
In[27]:= δ /: (δ^α_ _α_)_hp := d /; IndexQ[α]
         δ /: (δ^α_ _α_)_hp := 1 /; CoordinateQ[α]
         δ /: (δ^α_ _β_)_hp := 0 /; CoordinateQ[α] ∧ CoordinateQ[β]
```

The embedded derivatives, as well as the explicit derivatives, of the Kronecker delta are obviously zero. Also, since we are working in a metric connection, the covariant derivatives of the metric are zero [81, 326, 333].

```
In[30]:= g /: (g^α_ _β_ , ℓ__)_hp := 0
         δ /: (δ _α_ _β_ , ℓ__)_hp := 0
         g /: (g _α_ _β_ ; ℓ__)_hp := 0
         δ /: (δ _α_ _β_ ; ℓ__)_hp := 0
         ∂̄_α δ _α_ _β_ := 0
```

The metric tensor will lower any index in any tensor, as long as the index does not occur within a partial derivative. More generally, the Kronecker delta will raise and lower *any* index.

```
In[35]:= Contract ↗ Tensor /: (g^α_ _β_ T_ ℓ___ α_ r___)_hp :=
           T ℓ β r /; (",") ∉ {r}) ∧ IndexQ[α]
         Contract ↗ Tensor /: (g _α_ _β_ T_ ℓ___ ^α_ r___)_hp :=
           T ℓ β r /; (",") ∉ {r}) ∧ IndexQ[α]
         Contract ↗ Tensor /: (g _α_ ^β_ T_ ℓ___ β_ r___)_hp :=
           T ℓ α r /; (",") ∉ {r}) ∧ IndexQ[β]
         Contract ↗ Tensor /: (g _α_ _β_ T_ ℓ___ ^β_ r___)_hp :=
           T ℓ α r /; (",") ∉ {r}) ∧ IndexQ[β]
         Contract ↗ Tensor /: (δ^α_ _β_ T_ ℓ___ α_ r___)_hp := T ℓ β r /; IndexQ[α]
```

```
Contract ↗ Tensor /: (δ_{α_ β_} T_{ℓ___ α_}^{r___})_{hp} := T_{ℓ^ β^ r^} /; IndexQ[α]

Contract ↗ Tensor /: (δ_{α_}^{β_} T_{ℓ___ β_}^{r___})_{hp} := T_{ℓ^ α^ r^} /; IndexQ[β]

Contract ↗ Tensor /: (δ_{α_ β_} T_{ℓ___}^{β_ r___})_{hp} := T_{ℓ^ α^ r^} /; IndexQ[β]

Contract[any_] := DynamicBar @ any
```

Further, let us again combine the resolution operations.

```
In[44]:= Assign[Values @ ResolveΓ +{} Values @ ResolveR /. {
              ResolveΓ → Resolve,
              ResolveR → Resolve}]
```

Also, we would like our simplification to have the "contraction" rules, so we inherit them.

```
In[45]:= Assign[Values @ Contract /. Contract → Simplify]

         TagSetDelayed::write : Tag Simplify in Simplify[any_] is Protected.
```

Let us now perform some calculations from general relativity with these structures.

7.3.2 Linear Symmetries Revisited

In this subsection we revisit the linear symmetries previously commented on.

Let us re-verify the first Bianchi identity, which we previously showed in §6.14.1 *Origins of Linear Symmetries*. Only this time we use our new tensor handling mechanisms.

```
In[46]:= ResolveR [R_{αβγδ} + R_{αγδβ} + R_{αδβγ}]

Out[46]= -g_{αζ} Γ^{ϵ}_{γδ} Γ^{ζ}_{ϵβ} + g_{αζ} Γ^{ϵ}_{δγ} Γ^{ζ}_{ϵβ} + g_{αζ} Γ^{ϵ}_{βδ} Γ^{ζ}_{ϵγ} - g_{αζ} Γ^{ϵ}_{δβ} Γ^{ζ}_{ϵγ} -
         g_{αζ} Γ^{ϵ}_{βγ} Γ^{ζ}_{ϵδ} + g_{αζ} Γ^{ϵ}_{γβ} Γ^{ζ}_{ϵδ} - g_{αϵ} Γ^{ϵ}_{βγ,δ} + g_{αϵ} Γ^{ϵ}_{βδ,γ} +
         g_{αϵ} Γ^{ϵ}_{γβ,δ} - g_{αϵ} Γ^{ϵ}_{γδ,β} - g_{αϵ} Γ^{ϵ}_{δβ,γ} + g_{αϵ} Γ^{ϵ}_{δγ,β}

In[47]:= Canonicalize @ %

Out[47]= 0
```

Let us now verify the second Bianchi identity, which we previously only stated.

```
In[48]:= Canonicalize @ Act @
             ResolveR @ covariantsToPartials [R_{αβγδ;σ} + R_{αβδσ;γ} + R_{αβσγ;δ}]
```

```
Out[48]= -g[ϵ α, σ] Γ[ϵ, ζ δ] Γ[ζ, β γ] + g[ϵ α, δ] Γ[ϵ, ζ σ] Γ[ζ, β γ] - g[ϵ α, γ] Γ[ϵ, ζ σ] Γ[ζ, β δ] +
    g^(ϵ ζ) Γ[η, β γ] Γ[ϵ η δ] Γ[ζ α σ] + g[ϵ α, σ] Γ[ϵ ζ, γ] Γ[ζ β δ] - g[ϵ α, δ] Γ[ϵ ζ, γ] Γ[ζ β σ] +
    g[ϵ α, γ] Γ[ϵ ζ, δ] Γ[ζ β σ] + g^(ϵ ζ) Γ[η, β δ] Γ[ϵ α γ] Γ[ζ η σ] - g^(ϵ ζ) Γ[η, β γ] Γ[ϵ α δ] Γ[ζ η σ] -
    g^(ϵ ζ) Γ[ϵ, η γ] Γ[ζ α σ] Γ[η β δ] - g^(ϵ ζ) Γ[ϵ α γ] Γ[ζ, η δ] Γ[η β σ] + g^(ϵ ζ) Γ[ϵ, η γ] Γ[ζ α δ] Γ[η β σ] -
    g[ϵ α, σ] Γ[ϵ, β γ, δ] + g[ϵ α, δ] Γ[ϵ, β γ, σ] + g[ϵ α, σ] Γ[ϵ, β δ, γ] -
    g[ϵ α, γ] Γ[ϵ, β δ, σ] - g[ϵ α, δ] Γ[ϵ, β σ, γ] + g[ϵ α, γ] Γ[ϵ, β σ, δ] +
    Γ[ζ, β γ] Γ[η, ζ δ] Γ[ϵ η σ] δ[ϵ, α] + Γ[η, β δ] Γ[ϵ, ζ γ] Γ[ζ η σ] δ[ϵ, α] - Γ[ζ η, γ] Γ[ϵ ζ σ] Γ[η β δ] δ[ϵ, α] +
    Γ[ζ η, γ] Γ[ϵ ζ δ] Γ[η β σ] δ[ϵ, α] - Γ[ϵ, ζ γ] Γ[ζ, η δ] Γ[η β σ] δ[ϵ, α] - Γ[ζ, β γ] Γ[ϵ, η δ] Γ[η ζ σ] δ[ϵ, α] +
    Γ[ϵ ζ σ] Γ[ζ, β γ, δ] δ[ϵ, α] - Γ[ϵ ζ δ] Γ[ζ, β γ, σ] δ[ϵ, α] - Γ[ϵ ζ σ] Γ[ζ, β δ, γ] δ[ϵ, α] +
    Γ[ϵ ζ γ] Γ[ζ, β δ, σ] δ[ϵ, α] + Γ[ϵ ζ δ] Γ[ζ, β σ, γ] δ[ϵ, α] - Γ[ϵ ζ γ] Γ[ζ, β σ, δ] δ[ϵ, α] +
    Γ[ϵ α σ] Γ[ζ, β γ, δ] δ[ϵ, ζ] - Γ[ϵ α δ] Γ[ζ, β γ, σ] δ[ϵ, ζ] - Γ[ϵ α σ] Γ[ζ, β δ, γ] δ[ϵ, ζ] +
    Γ[ϵ α γ] Γ[ζ, β δ, σ] δ[ϵ, ζ] + Γ[ϵ α δ] Γ[ζ, β σ, γ] δ[ϵ, ζ] - Γ[ϵ α γ] Γ[ζ, β σ, δ] δ[ϵ, ζ]
```

Let us evaluate this expression in geodesic coordinates. In these coordinates the metric is *locally* flat, that is, the first partial derivatives of the metric are 0 at a given point. In addition, the Christoffel symbols are then also zero at that point. We can always transform to such a coordinate system — see [81, 326, 333]. In this locally flat frame, the expansion reduces to zero.

```
In[49]:= % /. {g[α_, β_, γ_] → 0, Γ[α_, β_, γ_] → 0}

Out[49]= 0
```

The equation is manifestly generally covariant, so since it holds in a locally flat metric, it holds in general [81, 326, 333].

Actually, it is heartening to see that all the index handling is working in an "automatic" way. Consider a term like $\bar{\partial}_\delta \Gamma_{\alpha\beta\gamma}$. Under Γ-resolution this becomes $\bar{\partial}_\delta \left(g_{\alpha\mu} \Gamma^{\mu}_{\ \beta\gamma}\right)$, which in turn is expanded. This all happens in the correct order and the right way through specifying the various actions under the various environments. That is, we do not have to make sure we apply one set of rules before another, etc. Everything just occurs in the correct order, since replacement and actions are occurring according to the normal *Mathematica* evaluation model. In fact, let us introduce a simple normal form to see this explicitly occurring.

```
In[50]:= Assign[Values @ ResolveΓ /.
            ResolveΓ → NormalFormΓ]
```

We override the specific rule for the expansion of the Christoffel symbols.

```
In[51]:= NormalFormΓ ↗ Γ /: Γ[λ_, α_, β_] =.
      NormalFormΓ @ any_ := DynamicBar @ ReIndex @ implicitizePD @ any
```

We see the index is automatically raised, and we obtain the derivative of the metric as required, all "automatically".

```
In[53]:= NormalFormΓ @ Γ[α, β, γ, δ]
```

$$\text{Out[53]=}\ g_{\alpha\epsilon,\delta}\,\Gamma^{\epsilon}{}_{\beta\gamma} + g_{\alpha\epsilon}\,\Gamma^{\epsilon}{}_{\beta\gamma,\delta}$$

Let us move on to an example which is more challenging to our canonicalization algorithm.

7.3.3 The Weyl Tensor and its Contractions

Let us now introduce the Weyl tensor, which we use to illustrate our canonicalization algorithm. Typically, this tensor is denoted by C, but we will use a script 'C' to distinguish it from the arbitrary constants that arise in solving differential equations in *Mathematica*. This tensor is used in conformal transformations [81, 326, 333]. The Weyl tensor shares the symmetries of the Riemann tensor. If we denote the dimensionality of our space by d, then for $d \geqslant 3$ the Weyl tensor can be decomposed into products of Riemann, Ricci, and metric tensors.

```
In[54]:= ResolveC ↗ C /: C[α_, β_, γ_, δ_] := ((-g[α, δ] g[β, γ] + g[α, γ] g[β, δ]) R)/((-2 + d) (d - 1)) - 1/(d - 2)
          (g[β, δ] R[α, γ] - g[β, γ] R[α, δ] - g[α, δ] R[β, γ] - g[α, γ] R[β, δ]) + R[α, β, γ, δ]
```

```
In[55]:= ResolveC @ any_ := ReIndex @ Expand @ any
```

Let us compute the complete contraction of the Weyl tensor.

```
In[56]:= C^{α, β, γ, δ} C[α, β, γ, δ]
```

$$\text{Out[56]=}\ C^{\alpha\beta\gamma\delta}\,C_{\alpha\beta\gamma\delta}$$

```
In[57]:= ResolveC @ %
```

Out[57]=

$$\frac{g^{\alpha\delta}\,g^{\beta\gamma}\,g_{\alpha\delta}\,g_{\beta\gamma}\,R^2}{(-2+d)^2\,(-1+d)^2}-\frac{g^{\alpha\gamma}\,g^{\beta\delta}\,g_{\alpha\delta}\,g_{\beta\gamma}\,R^2}{(-2+d)^2\,(-1+d)^2}-\frac{g^{\alpha\delta}\,g^{\beta\gamma}\,g_{\alpha\gamma}\,g_{\beta\delta}\,R^2}{(-2+d)^2\,(-1+d)^2}+$$
$$\frac{g^{\alpha\gamma}\,g^{\beta\delta}\,g_{\alpha\gamma}\,g_{\beta\delta}\,R^2}{(-2+d)^2\,(-1+d)^2}+\frac{g^{\beta\delta}\,g_{\alpha\delta}\,g_{\beta\gamma}\,R\,R^{\alpha\gamma}}{(-2+d)^2\,(-1+d)}-\frac{g^{\beta\delta}\,g_{\alpha\gamma}\,g_{\beta\delta}\,R\,R^{\alpha\gamma}}{(-2+d)^2\,(-1+d)}-$$
$$\frac{g^{\beta\gamma}\,g_{\alpha\delta}\,g_{\beta\gamma}\,R\,R^{\alpha\delta}}{(-2+d)^2\,(-1+d)}+\frac{g^{\beta\gamma}\,g_{\alpha\gamma}\,g_{\beta\delta}\,R\,R^{\alpha\delta}}{(-2+d)^2\,(-1+d)}-\frac{g^{\alpha\delta}\,g_{\alpha\delta}\,g_{\beta\gamma}\,R\,R^{\beta\gamma}}{(-2+d)^2\,(-1+d)}+$$
$$\frac{g^{\alpha\delta}\,g_{\alpha\gamma}\,g_{\beta\delta}\,R\,R^{\beta\gamma}}{(-2+d)^2\,(-1+d)}-\frac{g^{\alpha\gamma}\,g_{\alpha\delta}\,g_{\beta\gamma}\,R\,R^{\beta\delta}}{(-2+d)^2\,(-1+d)}+\frac{g^{\alpha\gamma}\,g_{\alpha\gamma}\,g_{\beta\delta}\,R\,R^{\beta\delta}}{(-2+d)^2\,(-1+d)}+$$
$$\frac{g^{\alpha\delta}\,g^{\beta\gamma}\,g_{\beta\delta}\,R\,R_{\alpha\gamma}}{(-2+d)^2\,(-1+d)}-\frac{g^{\alpha\gamma}\,g^{\beta\delta}\,g_{\beta\delta}\,R\,R_{\alpha\gamma}}{(-2+d)^2\,(-1+d)}+\frac{g^{\beta\delta}\,g_{\beta\delta}\,R^{\alpha\gamma}\,R_{\alpha\gamma}}{(-2+d)^2}-$$
$$\frac{g^{\beta\gamma}\,g_{\beta\delta}\,R^{\alpha\delta}\,R_{\alpha\gamma}}{(-2+d)^2}-\frac{g^{\alpha\delta}\,g_{\beta\delta}\,R^{\beta\gamma}\,R_{\alpha\gamma}}{(-2+d)^2}-\frac{g^{\alpha\gamma}\,g_{\beta\delta}\,R^{\beta\delta}\,R_{\alpha\gamma}}{(-2+d)^2}-$$
$$\frac{g^{\alpha\delta}\,g^{\beta\gamma}\,g_{\beta\gamma}\,R\,R_{\alpha\delta}}{(-2+d)^2\,(-1+d)}+\frac{g^{\alpha\gamma}\,g^{\beta\delta}\,g_{\beta\gamma}\,R\,R_{\alpha\delta}}{(-2+d)^2\,(-1+d)}-\frac{g^{\beta\delta}\,g_{\beta\gamma}\,R^{\alpha\gamma}\,R_{\alpha\delta}}{(-2+d)^2}+$$
$$\frac{g^{\beta\gamma}\,g_{\beta\gamma}\,R^{\alpha\delta}\,R_{\alpha\delta}}{(-2+d)^2}+\frac{g^{\alpha\delta}\,g_{\beta\gamma}\,R^{\beta\gamma}\,R_{\alpha\delta}}{(-2+d)^2}+\frac{g^{\alpha\gamma}\,g_{\beta\gamma}\,R^{\beta\delta}\,R_{\alpha\delta}}{(-2+d)^2}-$$
$$\frac{g^{\alpha\delta}\,g^{\beta\gamma}\,g_{\alpha\delta}\,R\,R_{\beta\gamma}}{(-2+d)^2\,(-1+d)}+\frac{g^{\alpha\gamma}\,g^{\beta\delta}\,g_{\alpha\delta}\,R\,R_{\beta\gamma}}{(-2+d)^2\,(-1+d)}-\frac{g^{\beta\delta}\,g_{\alpha\delta}\,R^{\alpha\gamma}\,R_{\beta\gamma}}{(-2+d)^2}+$$
$$\frac{g^{\beta\gamma}\,g_{\alpha\delta}\,R^{\alpha\delta}\,R_{\beta\gamma}}{(-2+d)^2}+\frac{g^{\alpha\delta}\,g_{\alpha\delta}\,R^{\beta\gamma}\,R_{\beta\gamma}}{(-2+d)^2}+\frac{g^{\alpha\gamma}\,g_{\alpha\delta}\,R^{\beta\delta}\,R_{\beta\gamma}}{(-2+d)^2}-$$
$$\frac{g^{\alpha\delta}\,g^{\beta\gamma}\,g_{\alpha\gamma}\,R\,R_{\beta\delta}}{(-2+d)^2\,(-1+d)}+\frac{g^{\alpha\gamma}\,g^{\beta\delta}\,g_{\alpha\gamma}\,R\,R_{\beta\delta}}{(-2+d)^2\,(-1+d)}-\frac{g^{\beta\delta}\,g_{\alpha\gamma}\,R^{\alpha\gamma}\,R_{\beta\delta}}{(-2+d)^2}+$$
$$\frac{g^{\beta\gamma}\,g_{\alpha\gamma}\,R^{\alpha\delta}\,R_{\beta\delta}}{(-2+d)^2}+\frac{g^{\alpha\delta}\,g_{\alpha\gamma}\,R^{\beta\gamma}\,R_{\beta\delta}}{(-2+d)^2}+\frac{g^{\alpha\gamma}\,g_{\alpha\gamma}\,R^{\beta\delta}\,R_{\beta\delta}}{(-2+d)^2}-$$
$$\frac{g_{\alpha\delta}\,g_{\beta\gamma}\,R\,R^{\alpha\beta\gamma\delta}}{(-2+d)\,(-1+d)}+\frac{g_{\alpha\gamma}\,g_{\beta\delta}\,R\,R^{\alpha\beta\gamma\delta}}{(-2+d)\,(-1+d)}-\frac{g_{\beta\delta}\,R_{\alpha\gamma}\,R^{\alpha\beta\gamma\delta}}{-2+d}+$$
$$\frac{g_{\beta\gamma}\,R_{\alpha\delta}\,R^{\alpha\beta\gamma\delta}}{-2+d}+\frac{g_{\alpha\delta}\,R_{\beta\gamma}\,R^{\alpha\beta\gamma\delta}}{-2+d}+\frac{g_{\alpha\gamma}\,R_{\beta\delta}\,R^{\alpha\beta\gamma\delta}}{-2+d}-$$
$$\frac{g^{\alpha\delta}\,g^{\beta\gamma}\,R\,R_{\alpha\beta\gamma\delta}}{(-2+d)\,(-1+d)}+\frac{g^{\alpha\gamma}\,g^{\beta\delta}\,R\,R_{\alpha\beta\gamma\delta}}{(-2+d)\,(-1+d)}-\frac{g^{\beta\delta}\,R^{\alpha\gamma}\,R_{\alpha\beta\gamma\delta}}{-2+d}+$$
$$\frac{g^{\beta\gamma}\,R^{\alpha\delta}\,R_{\alpha\beta\gamma\delta}}{-2+d}+\frac{g^{\alpha\delta}\,R^{\beta\gamma}\,R_{\alpha\beta\gamma\delta}}{-2+d}+\frac{g^{\alpha\gamma}\,R^{\beta\delta}\,R_{\alpha\beta\gamma\delta}}{-2+d}+R^{\alpha\beta\gamma\delta}\,R_{\alpha\beta\gamma\delta}$$

In[58]:= `Contract@ %`

```
In[58]:= 
```

$$\text{Out[58]}= -\frac{2\,d\,\mathbf{R}^2}{(-2+d)^2\,(-1+d)^2}+\frac{2\,d^2\,\mathbf{R}^2}{(-2+d)^2\,(-1+d)^2}-\frac{2\,d\,\mathbf{R}\,\mathbf{R}^{\gamma}{}_{\gamma}}{(-2+d)^2\,(-1+d)}+\frac{2\,\mathbf{R}\,\mathbf{R}^{\delta}{}_{\delta}}{(-2+d)^2\,(-1+d)}-\frac{\mathbf{R}^{\alpha\gamma}\,\mathbf{R}_{\alpha\gamma}}{(-2+d)^2}+\frac{d\,\mathbf{R}^{\alpha\gamma}\,\mathbf{R}_{\alpha\gamma}}{(-2+d)^2}-\frac{\mathbf{R}^{\alpha\delta}\,\mathbf{R}_{\alpha\delta}}{(-2+d)^2}+\frac{d\,\mathbf{R}^{\alpha\delta}\,\mathbf{R}_{\alpha\delta}}{(-2+d)^2}+\frac{\mathbf{R}^{\beta\gamma}\,\mathbf{R}_{\beta\gamma}}{(-2+d)^2}+\frac{d\,\mathbf{R}^{\beta\gamma}\,\mathbf{R}_{\beta\gamma}}{(-2+d)^2}+\frac{\mathbf{R}^{\beta\delta}\,\mathbf{R}_{\beta\delta}}{(-2+d)^2}+\frac{d\,\mathbf{R}^{\beta\delta}\,\mathbf{R}_{\beta\delta}}{(-2+d)^2}+\frac{2\,\mathbf{R}\,\mathbf{R}_{\gamma}{}^{\gamma}}{(-2+d)^2\,(-1+d)}-\frac{2\,d\,\mathbf{R}\,\mathbf{R}_{\gamma}{}^{\gamma}}{(-2+d)^2\,(-1+d)}+\frac{2\,\mathbf{R}^{\gamma}{}_{\delta}\,\mathbf{R}_{\gamma}{}^{\delta}}{(-2+d)^2}-\frac{2\,\mathbf{R}^{\delta}{}_{\gamma}\,\mathbf{R}_{\delta}{}^{\gamma}}{(-2+d)^2}+\frac{\mathbf{R}_{\alpha\delta}\,\mathbf{R}^{\alpha\ \gamma\delta}_{\ \gamma}}{-2+d}-\frac{\mathbf{R}_{\alpha\gamma}\,\mathbf{R}^{\alpha\ \gamma\delta}_{\ \delta}}{-2+d}+\frac{\mathbf{R}\,\mathbf{R}^{\gamma\delta}{}_{\gamma\delta}}{(-2+d)\,(-1+d)}+\frac{\mathbf{R}^{\beta\delta}\,\mathbf{R}^{\gamma}{}_{\beta\gamma\delta}}{-2+d}-\frac{\mathbf{R}\,\mathbf{R}^{\delta\gamma}{}_{\gamma\delta}}{(-2+d)\,(-1+d)}+\frac{\mathbf{R}^{\beta\gamma}\,\mathbf{R}^{\delta}{}_{\beta\gamma\delta}}{-2+d}+\frac{\mathbf{R}^{\alpha\delta}\,\mathbf{R}_{\alpha\ \gamma\delta}^{\ \gamma}}{-2+d}-\frac{\mathbf{R}^{\alpha\gamma}\,\mathbf{R}_{\alpha\ \gamma\delta}^{\ \delta}}{-2+d}+\mathbf{R}^{\alpha\beta\gamma\delta}\,\mathbf{R}_{\alpha\beta\gamma\delta}+\frac{\mathbf{R}_{\beta\delta}\,\mathbf{R}_{\gamma}{}^{\beta\gamma\delta}}{-2+d}+\frac{\mathbf{R}\,\mathbf{R}_{\gamma\delta}{}^{\gamma\delta}}{(-2+d)\,(-1+d)}+\frac{\mathbf{R}_{\beta\gamma}\,\mathbf{R}_{\delta}{}^{\beta\gamma\delta}}{-2+d}-\frac{\mathbf{R}\,\mathbf{R}_{\delta\gamma}{}^{\gamma\delta}}{(-2+d)\,(-1+d)}$$

```
In[59]:= Canonicalize @ %
```

$$\text{Out[59]}= \frac{2\,d\,\mathbf{R}^2}{(-2+d)^2\,(-1+d)}+\frac{4\,\mathbf{R}\,\mathbf{R}^{\alpha}{}_{\alpha}}{(-2+d)^2\,(-1+d)}-\frac{4\,d\,\mathbf{R}\,\mathbf{R}^{\alpha}{}_{\alpha}}{(-2+d)^2\,(-1+d)}-\frac{4\,d\,\mathbf{R}^{\alpha\beta}\,\mathbf{R}_{\alpha\beta}}{(-2+d)^2\,(-1+d)}+\frac{4\,d^2\,\mathbf{R}^{\alpha\beta}\,\mathbf{R}_{\alpha\beta}}{(-2+d)^2\,(-1+d)}-\frac{8\,\mathbf{R}\,\mathbf{R}^{\alpha\beta}{}_{\alpha\beta}}{(-2+d)^2\,(-1+d)}+\frac{4\,d\,\mathbf{R}\,\mathbf{R}^{\alpha\beta}{}_{\alpha\beta}}{(-2+d)^2\,(-1+d)}+\frac{8\,\mathbf{R}^{\alpha\beta}\,\mathbf{R}^{\gamma}{}_{\alpha\beta\gamma}}{(-2+d)^2\,(-1+d)}-\frac{12\,d\,\mathbf{R}^{\alpha\beta}\,\mathbf{R}^{\gamma}{}_{\alpha\beta\gamma}}{(-2+d)^2\,(-1+d)}+\frac{4\,d^2\,\mathbf{R}^{\alpha\beta}\,\mathbf{R}^{\gamma}{}_{\alpha\beta\gamma}}{(-2+d)^2\,(-1+d)}-\frac{4\,\mathbf{R}^{\alpha\beta\gamma\delta}\,\mathbf{R}_{\alpha\beta\gamma\delta}}{(-2+d)^2\,(-1+d)}+\frac{8\,d\,\mathbf{R}^{\alpha\beta\gamma\delta}\,\mathbf{R}_{\alpha\beta\gamma\delta}}{(-2+d)^2\,(-1+d)}-\frac{5\,d^2\,\mathbf{R}^{\alpha\beta\gamma\delta}\,\mathbf{R}_{\alpha\beta\gamma\delta}}{(-2+d)^2\,(-1+d)}+\frac{d^3\,\mathbf{R}^{\alpha\beta\gamma\delta}\,\mathbf{R}_{\alpha\beta\gamma\delta}}{(-2+d)^2\,(-1+d)}$$

```
In[60]:= Simplify /@ %
```

$$\text{Out[60]}= -\frac{4\,\mathbf{R}^2}{(-2+d)^2\,(-1+d)}+\frac{2\,d\,\mathbf{R}^2}{(-2+d)^2\,(-1+d)}-\frac{8\,\mathbf{R}^{\alpha\beta}\,\mathbf{R}_{\alpha\beta}}{(-2+d)^2\,(-1+d)}+\frac{8\,d\,\mathbf{R}^{\alpha\beta}\,\mathbf{R}_{\alpha\beta}}{(-2+d)^2\,(-1+d)}-\frac{4\,\mathbf{R}^{\alpha\beta\gamma\delta}\,\mathbf{R}_{\alpha\beta\gamma\delta}}{(-2+d)^2\,(-1+d)}+\frac{8\,d\,\mathbf{R}^{\alpha\beta\gamma\delta}\,\mathbf{R}_{\alpha\beta\gamma\delta}}{(-2+d)^2\,(-1+d)}-\frac{5\,d^2\,\mathbf{R}^{\alpha\beta\gamma\delta}\,\mathbf{R}_{\alpha\beta\gamma\delta}}{(-2+d)^2\,(-1+d)}+\frac{d^3\,\mathbf{R}^{\alpha\beta\gamma\delta}\,\mathbf{R}_{\alpha\beta\gamma\delta}}{(-2+d)^2\,(-1+d)}$$

In four dimensions this simplifies to the following.

```
In[61]:= % /. d → 4
```

$$\text{Out[61]=}\quad \frac{\mathbf{R}^2}{3} + 2\,\mathbf{R}^{\alpha\beta}\,\mathbf{R}_{\alpha\beta} + \mathbf{R}^{\alpha\beta\gamma\delta}\,\mathbf{R}_{\alpha\beta\gamma\delta}$$

This particular contraction is used in conformal transformations. It is also used in the Petrov classification scheme [209, 211, 212], whereby after calculating certain invariants, one can determine if a given metric is unique, or if indeed one has only found a transformed form of an already known metric. For a more advanced challenge, let us calculate a higher order contraction, $C^{\alpha\beta\gamma\delta}\,C^{\epsilon\zeta}{}_{\alpha\beta}\,C_{\gamma\delta\epsilon\zeta}$. Unfortunately, when our calculations start yielding thousands of terms consisting of complex tensor products, we can no longer just "whack" the expression with `Canonicalize`. Instead, we have to be a bit more intelligent in our application of our tools. To this end, in order to calculate the triple contraction of the Weyl tensor, let us proceed as follows. First, we remove the reindexing from the resolution of the Weyl tensor, since this can take a long time if there are a large number of terms.

```
In[62]:= ResolveC @ any_ := Expand @ any
```

Then we resolve the product, as done previously.

```
In[63]:= invariant = ResolveC[C^αβγδ C^ϵζ_αβ C_γδϵζ]; // Timing
```

```
Out[63]= {0.0666667 Second, Null}
```

```
In[64]:= Count[invariant, _Tensor, {0, ∞}]
         Length @ invariant
```

```
Out[64]= 2129
```

```
Out[65]= 343
```

Thus, we have some 2000 tensors comprising 343 terms. By contracting metrics and also the resulting Kronecker deltas, we can eliminate a good portion of these terms.

```
In[66]:= inv1 = Contract@ invariant; // Timing
```

```
Out[66]= {0.633333 Second, Null}
```

```
In[67]:= Length @ inv1
```

```
Out[67]= 174
```

Also, to ensure that our timings are representative, let us clear the tensor caches.

```
In[68]:= ClearTensorCaches[]
```

We can now perform the canonicalization.

```
In[69]:= inv2 = Canonicalize @ inv1; // Timing
```

```
Out[69]= {16.5 Second, Null}
```

```
In[70]:= Length @ inv2
```

```
Out[70]= 33
```

In four dimensions, this simplifies as follows.

```
In[71]:= Simplify[inv2 /. d → 4]
```

$$\text{Out[71]=}\quad \frac{\mathbf{R}^3}{18} + \mathbf{R}\left(-\frac{1}{2}\mathbf{R}^{\alpha\beta}\mathbf{R}_{\alpha\beta} + \mathbf{R}^{\alpha\beta\gamma\delta}\mathbf{R}_{\alpha\beta\gamma\delta}\right) + \mathbf{R}^{\alpha\beta}\left(4\mathbf{R}^{\gamma}{}_{\alpha}\mathbf{R}_{\beta\gamma} - 6\mathbf{R}^{\gamma\delta\epsilon}{}_{\alpha}\mathbf{R}_{\beta\gamma\delta\epsilon}\right) + \mathbf{R}^{\alpha\beta\gamma\delta}\mathbf{R}^{\epsilon\zeta}{}_{\alpha\beta}\mathbf{R}_{\gamma\delta\epsilon\zeta}$$

Consequently, we see that it is easily viable to do these larger calculations involving several thousand terms. We could easily calculate the other higher order non-vanishing curvature invariants.

Before we close this subsection, let us mention that we can also easily perform fairly complex calculations involving higher order products of the Riemann tensor. Indeed, here is a calculation that is quoted by MapleTensor. Unfortunately, they do not report the time their system took to achieve this result.

```
In[72]:= Canonicalize[R^d_acb R^m_j^io R^cj_d m R^ba_io + R^cj_d m R^dab_c R^iom_j R_ioab -
           R^ia_o b R^jo_m i R^bc_d a R^md_j c] // Timing
```

```
Out[72]= {34.8167 Second, 0}
```

Note that since we have solved this problem, we have in some sense solved all equivalent problems that are related by the symmetries to this expression. Also note that if we encoded our algorithm in C++, then we would likely be able to speed it up by around 500 times.

7.3.4 Yang-Mills Fields

To compare the style of calculation in our system to that of MathTensor, let us perform a computation which is also presented in Parker and Christensen[258]. The calculation expands the totally contracted field term in the Yang-Mills Lagrangian.

The Yang-Mills field resolves in terms of the gauge fields as follows.

```
In[73]:= ResolveYM ↗ 𝓕 /: 𝓕^i_ α_ β_ := 𝒜^i α*, β* - 𝒜^i β*, α* + g_ym f^i j k 𝒜^j α* 𝒜^k β*
         ResolveYM @ any_ := DynamicBar @ any
```

Technical Note: We have used the vector potential $\mathcal{A}$ here as opposed to A, since the latter has the symmetries of the totally anti-symmetric tensor. We could introduce further sophistication into our algorithms so that our symmetries are index class dependent. However, since we sometimes desire that the symmetries remain the same, say in a 1+3 split, it is probably easier just to use a different tensor name.

We can now resolve the completely contracted field term which arrises in the Yang-Mills Lagrangian.

```
In[75]:= 𝓕^i α β 𝓕_i^α β
```

```
In[75]:= 
```

$$\text{Out[75]=}\ \mathcal{F}^{i}{}_{\alpha\beta}\,\mathcal{F}_{i}{}^{\alpha\beta}$$

```
In[76]:= ReIndex @ Expand @ ResolveYM @ %
```

$$\begin{aligned}\text{Out[76]=}\ & \mathcal{A}^{i}{}_{\alpha,\beta}\,\mathcal{A}_{i}{}^{\alpha\ \beta}{}_{,} - \mathcal{A}^{i}{}_{\beta,\alpha}\,\mathcal{A}_{i}{}^{\alpha\ \beta}{}_{,} - \mathcal{A}^{i}{}_{\alpha,\beta}\,\mathcal{A}_{i}{}^{\beta\ \alpha}{}_{,} + \\ & \mathcal{A}^{i}{}_{\beta,\alpha}\,\mathcal{A}_{i}{}^{\beta\ \alpha}{}_{,} + g_{ym}\,\mathcal{A}^{i1}{}_{\alpha}\,\mathcal{A}^{j}{}_{\beta}\,\mathcal{A}_{i}{}^{\alpha\ \beta}{}_{,}\,f^{i}{}_{i1\,j} - \\ & g_{ym}\,\mathcal{A}^{i1}{}_{\alpha}\,\mathcal{A}^{j}{}_{\beta}\,\mathcal{A}_{i}{}^{\beta\ \alpha}{}_{,}\,f^{i}{}_{i1\,j} + g_{ym}\,\mathcal{A}^{i1\,\alpha}\,\mathcal{A}^{j\,\beta}\,\mathcal{A}^{l}{}_{\alpha,\beta}\,f_{i\,i1\,j} - \\ & g_{ym}\,\mathcal{A}^{i1\,\alpha}\,\mathcal{A}^{j\,\beta}\,\mathcal{A}^{i}{}_{\beta,\alpha}\,f_{i\,i1\,j} + g_{ym}^{2}\,\mathcal{A}^{i1\,\beta}\,\mathcal{A}^{i2\,\alpha}\,\mathcal{A}^{j}{}_{\beta}\,\mathcal{A}^{j1}{}_{\alpha}\,f^{i}{}_{j1\,j}\,f_{i\,i2\,i1}\end{aligned}$$

Let us declare that a few more symbols are `SU2StructureIndices` in order to make our calculations more readable.

```
In[77]:= DeclareIndexClass[SU2StructureIndices, {i, j, k, l, m, n}];
```

The structure factors are completely anti-symmetric and in addition, products of the structure factors are resolvable into products of the Kronecker deltas.

```
In[78]:= DeclareSymmetries[f, 3, {(1 ↔ 2)_, (2 ↔ 3)_}]
         ResolveSU2 ↗ Tensor /: f_i_j_k_ f^i_l_m_ := δ_j_l δ_k_m - δ_j_m δ_k_l
```

The `ResolveSU2` should act like a resolution environment, so we inherit those rules.

```
In[80]:= Assign[Values @ Resolve /. Resolve → ResolveSU2]
```

```
In[81]:= ResolveSU2 @ ReIndex @ Expand @ ResolveYM [ F^i_αβ F_i^αβ ]
```

$$\begin{aligned}\text{Out[81]=}\ & -g_{ym}^{2}\,\mathcal{A}^{i}{}_{\beta}\,\mathcal{A}^{j\,\beta}\,\mathcal{A}^{k}{}_{\alpha}\,\mathcal{A}^{l\,\alpha}\,\delta_{j\,k}\,\delta_{i\,l} + g_{ym}^{2}\,\mathcal{A}^{i}{}_{\beta}\,\mathcal{A}^{j\,\beta}\,\mathcal{A}^{k}{}_{\alpha}\,\mathcal{A}^{l\,\alpha}\,\delta_{j\,l}\,\delta_{i\,k} + \\ & g_{ym}\,\mathcal{A}^{j\,\beta}\,\mathcal{A}^{k\,\alpha}\,f_{i\,k\,j}\,\bar{\partial}_{\beta}\mathcal{A}^{i}{}_{\alpha} - g_{ym}\,\mathcal{A}^{j\,\beta}\,\mathcal{A}^{k\,\alpha}\,f_{i\,k\,j}\,\bar{\partial}_{\alpha}\mathcal{A}^{i}{}_{\beta} + \\ & g_{ym}\,\mathcal{A}^{j}{}_{\beta}\,\mathcal{A}^{k}{}_{\alpha}\,f^{i}{}_{k\,j}\,\bar{\partial}^{\beta}\mathcal{A}_{i}{}^{\alpha} + \bar{\partial}_{\beta}\mathcal{A}^{l}{}_{\alpha}\,\bar{\partial}^{\beta}\mathcal{A}_{i}{}^{\alpha} - \bar{\partial}_{\alpha}\mathcal{A}^{i}{}_{\beta}\,\bar{\partial}^{\beta}\mathcal{A}_{i}{}^{\alpha} - \\ & g_{ym}\,\mathcal{A}^{j}{}_{\beta}\,\mathcal{A}^{k}{}_{\alpha}\,f^{i}{}_{k\,j}\,\bar{\partial}^{\alpha}\mathcal{A}_{i}{}^{\beta} - \bar{\partial}_{\beta}\mathcal{A}^{i}{}_{\alpha}\,\bar{\partial}^{\alpha}\mathcal{A}_{i}{}^{\beta} + \bar{\partial}_{\alpha}\mathcal{A}^{i}{}_{\beta}\,\bar{\partial}^{\alpha}\mathcal{A}_{i}{}^{\beta}\end{aligned}$$

We are working in a locally flat metric, so we can raise and lower the indices on the gauge fields.

```
In[82]:= Canonicalize[%, MetricLocallyFlat → True]
```

$$\begin{aligned}\text{Out[82]=}\ & 2\,\mathcal{A}^{i\,\alpha\ \beta}{}_{,}\,\mathcal{A}_{i\,\alpha,\beta} - 2\,\mathcal{A}^{i\,\alpha\ \beta}{}_{,}\,\mathcal{A}_{i\,\beta,\alpha} + 4\,g_{ym}\,\mathcal{A}^{i\,\alpha}\,\mathcal{A}^{j\,\beta}\,\mathcal{A}^{k}{}_{\alpha,\beta}\,f_{i\,j\,k} + \\ & g_{ym}^{2}\,\mathcal{A}^{i\,\alpha}\,\mathcal{A}^{j\,\beta}\,\mathcal{A}^{k}{}_{\alpha}\,\mathcal{A}^{l}{}_{\beta}\,\delta_{i\,k}\,\delta_{j\,l} - g_{ym}^{2}\,\mathcal{A}^{i\,\alpha}\,\mathcal{A}^{j\,\beta}\,\mathcal{A}^{k}{}_{\alpha}\,\mathcal{A}^{l}{}_{\beta}\,\delta_{i\,j}\,\delta_{k\,l}\end{aligned}$$

As Parker & Christensen notes, this expansion leads to the cubic and quartic interactions in the Yang-Mills Lagrangian [138, 186, 278].

7.3.5 Component Calculations and the Schwarzschild Metric

Our main purpose here is to demonstrate our canonicalization algorithm and our language modifications. However, in some approaches, one works with components themselves as opposed to the abstract tensors. Let us consequently give a simple example involving a component calculation.

As is by now evident, we can easily resolve tensorial expressions into expressions involving just the metric and derivatives thereof. For instance, say we would like to calculate the Christoffel symbols of a given metric. We would expand the Christoffel symbols like so.

```
In[83]:= ResolveΓ @ Γ[α, β, γ]
```

$$\text{Out[83]=} \quad -\frac{1}{2}\,\mathbf{g}^{\alpha\delta}\,\bar{\partial}_{\delta}\mathbf{g}_{\beta\gamma} + \frac{1}{2}\,\mathbf{g}^{\alpha\delta}\,\bar{\partial}_{\gamma}\mathbf{g}_{\beta\delta} + \frac{1}{2}\,\mathbf{g}^{\alpha\delta}\,\bar{\partial}_{\beta}\mathbf{g}_{\delta\gamma}$$

Thus, if we needed to calculate say $\Gamma^{1}{}_{02}$, we would do so as follows.

```
In[84]:= ExpandContraction @ ResolveΓ @ Γ[1, 0, 2]
```

$$\text{Out[84]=} \quad \frac{1}{2}\,\mathbf{g}^{10}\,\bar{\partial}_{2}\mathbf{g}_{00} + \frac{1}{2}\,\mathbf{g}^{11}\,\bar{\partial}_{2}\mathbf{g}_{01} - \frac{1}{2}\,\mathbf{g}^{11}\,\bar{\partial}_{1}\mathbf{g}_{02} - \frac{1}{2}\,\mathbf{g}^{13}\,\bar{\partial}_{3}\mathbf{g}_{02} + \frac{1}{2}\,\mathbf{g}^{13}\,\bar{\partial}_{2}\mathbf{g}_{03} + \frac{1}{2}\,\mathbf{g}^{11}\,\bar{\partial}_{0}\mathbf{g}_{12} + \frac{1}{2}\,\mathbf{g}^{12}\,\bar{\partial}_{0}\mathbf{g}_{22} + \frac{1}{2}\,\mathbf{g}^{13}\,\bar{\partial}_{0}\mathbf{g}_{32}$$

Of course, to evaluate these derivatives, we need to work in a specific metric. Let us perform this calculation for the Schwarzschild metric [81, 255, 326, 333]. The Schwarzschild metric can be stated as follows.

```
In[85]:= gMatrix = {{(1-2 m)/r, 0, 0, 0}, {0, r/(1-2 m), 0, 0}, {0, 0, r^2, 0}, {0, 0, 0, r^2 Sin[θ]^2}};
```

Let us introduce a function which substitutes the primitive values of the metric.

```
In[86]:= Table[PrimitiveForm↗g /: g[i, j] = gMatrix[[i + 1, j + 1]], {i, 0, 3}, {j, 0, 3}];
```

```
In[87]:= gInverse = Simplify @ Inverse @ gMatrix;
```

```
In[88]:= Table[PrimitiveForm↗g /: g^(i j) = gInverse[[i + 1, j + 1]], {i, 0, 3}, {j, 0, 3}];
```

We also need our `PrimitiveForm` operation to be able to resolve any tensorial expressions it encounters, so we inherit all the values of the resolution environment. We also want our partial derivatives to act on objects, so we inherit the values of `Act`.

```
In[89]:= Assign[Values @ Act /. Act → PrimitiveForm]
         Assign[Values @ Resolve /. Resolve → PrimitiveForm]
```

Our tensor partial derivatives act like "partial derivatives" on the semicommutative multipliers.

```
In[91]:= PrimitiveForm ↗ ∂̄0 expr_?$CMultiplierQ := ∂t expr
         PrimitiveForm ↗ ∂̄1 expr_?$CMultiplierQ := ∂r expr
         PrimitiveForm ↗ ∂̄2 expr_?$CMultiplierQ := ∂θ expr
         PrimitiveForm ↗ ∂̄3 expr_?$CMultiplierQ := ∂φ expr
```

```
In[95]:= DeclareSCMultiplier @ {t, r, m, θ, φ, Sin[val_ /; $CMultiplierQ @ val]}
```

Finally, if there are any dummy indices in the expression given to `PrimitiveForm`, then we need to contract over them.

```
In[96]:= PrimitiveForm @ expr_ :=
           PrimitiveForm @ ExpandContraction @ expr /; DummyIndices @ expr ≠ {}
```

Now we can simply apply `PrimitiveForm` to an expression in order to obtain its "primitive value".

```
In[97]:= PrimitiveForm @ Γ^1_{2 2}
```

```
Out[97]= -1 + 2 m
```

```
In[98]:= PrimitiveForm[g^{α β} g_{α β}]
```

```
Out[98]= 4
```

This operation works partially when it can.

```
In[99]:= PrimitiveForm @ Γ_{α β γ}
```

$$\text{Out[99]}= \frac{r\, g_{\alpha 0}\, \bar{\partial}_\beta g_{0\gamma}}{2\,(1-2m)} + \frac{g_{\alpha 1}\, \bar{\partial}_\beta g_{1\gamma}}{2r} - \frac{m\, g_{\alpha 1}\, \bar{\partial}_\beta g_{1\gamma}}{r} + \frac{g_{\alpha 2}\, \bar{\partial}_\beta g_{2\gamma}}{2r^2} + \frac{\mathrm{Csc}[\theta]^2\, g_{\alpha 3}\, \bar{\partial}_\beta g_{3\gamma}}{2r^2} + \frac{r\, g_{\alpha 0}\, \bar{\partial}_\gamma g_{\beta 0}}{2\,(1-2m)} + \frac{g_{\alpha 1}\, \bar{\partial}_\gamma g_{\beta 1}}{2r} - \frac{m\, g_{\alpha 1}\, \bar{\partial}_\gamma g_{\beta 1}}{r} + \frac{g_{\alpha 2}\, \bar{\partial}_\gamma g_{\beta 2}}{2r^2} + \frac{\mathrm{Csc}[\theta]^2\, g_{\alpha 3}\, \bar{\partial}_\gamma g_{\beta 3}}{2r^2} - \frac{r\, g_{\alpha 0}\, \bar{\partial}_0 g_{\beta\gamma}}{2\,(1-2m)} - \frac{g_{\alpha 1}\, \bar{\partial}_1 g_{\beta\gamma}}{2r} + \frac{m\, g_{\alpha 1}\, \bar{\partial}_1 g_{\beta\gamma}}{r} - \frac{g_{\alpha 2}\, \bar{\partial}_2 g_{\beta\gamma}}{2r^2} - \frac{\mathrm{Csc}[\theta]^2\, g_{\alpha 3}\, \bar{\partial}_3 g_{\beta\gamma}}{2r^2}$$

Here is the table of the standard form of the Christoffel symbols for the Schwarzschild metric.

```
In[100]:= MatrixForm /@ Simplify @
            PrimitiveForm @ Table[Γ^α_{β γ}, {α, 0, 3}, {β, 0, 3}, {γ, 0, 3}]
```

```
Out[100]= {{{0, -1/(2 r), 0, 0}, {-1/(2 r), 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}},
 {{(1-2 m)^2/(2 r^3), 0, 0, 0}, {0, 1/(2 r), 0, 0}, {0, 0, -1+2 m, 0},
  {0, 0, 0, (-1+2 m) Sin[θ]^2}},
 {{0, 0, 0, 0}, {0, 0, 1/r, 0}, {0, 1/r, 0, 0}, {0, 0, 0, -Cos[θ] Sin[θ]}},
 {{0, 0, 0, 0}, {0, 0, 0, 1/r}, {0, 0, 0, Cot[θ]}, {0, 1/r, Cot[θ], 0}}}
```

Thus we have obtained the Christoffel symbols for the Schwarzschild metric. Of course, if we were to apply our calculations to large expressions, it would be extremely wise to cache values, since there is a large amount of redundant calculations occurring. This style of calculation is done in other programs, and any further implementation here would not illustrate this point further. For instance, if we wanted to calculate the complete contraction of the Riemann tensor, we would simply perform the following.

```
In[101]:= Canonicalize @ ExpandContraction[ R^{α β γ δ} R_{α β γ δ} ]

Out[101]= 4 R^{0101} R_{0101} + 8 R^{0102} R_{0102} + 8 R^{0103} R_{0103} + 8 R^{0112} R_{0112} +
  8 R^{0113} R_{0113} + 16 R^{0123} R_{0123} - 8 R^{0213} R_{0123} +
  4 R^{0202} R_{0202} + 8 R^{0203} R_{0203} + 8 R^{0212} R_{0212} - 8 R^{0123} R_{0213} +
  16 R^{0213} R_{0213} + 8 R^{0223} R_{0223} + 4 R^{0303} R_{0303} +
  8 R^{0313} R_{0313} + 8 R^{0323} R_{0323} + 4 R^{1212} R_{1212} + 8 R^{1213} R_{1213} +
  8 R^{1223} R_{1223} + 4 R^{1313} R_{1313} + 8 R^{1323} R_{1323} + 4 R^{2323} R_{2323}
```

Then we would plug in the various components. There is nothing intrinsically difficult about such calculations, and it should be clear how to proceed. Consequently, we omit further calculations and move on to the last example, that of quasi-spin.

7.4 Example: Quasi-Spin

7.4.1 Introduction to the Problem

In §5 *Prototypical Structures and Quantum Mechanics*, we performed some calculations involving angular momentum. Let us now tackle a more advanced problem, that of the quasi-spin commutation relations. We would like to reproduce and correct the calculations of Savage and Stedman. Savage's thesis is concerned with "Higher Symmetries in Jahn-Teller systems". He performs many calculations involving quasi-spin, its behavior and its interaction with other operations. The second quantized commutation relations that arise are extremely cumbersome to perform by hand. So much so that Savage neglected to verify that his formulation of the quasi-spin operator was a spherical tensor-operator in quasi-spin space. It transpires that it is not. This arose because the Fano-Racah contra-standard transformation between Cartesian quasi-spin components and spherical quasi-spin components was used, as opposed to the more standard transformation. We will confirm that the Fano-Racah contra-standard transformation leads to the incorrect commutation relations for the quasi-spin operation, while the standard transformation leads to the correct result.

Again it is largely outside the purview of this thesis to enter into the theory behind such operations so we will content ourselves to verify and reproduce some of the calculations in the thesis of Savage[284]. These are concerned with verifying and deriving relations involving the quasi-spin operator, and also being able to manipulate expressions containing these operators as well as other operators which act on the quasi-spin operator, such as complex conjugation, time reversal, particle hole conjugation, tensor couplings, etc. To progress to these issues, we need to discuss second quantization in shell theory.

Unfortunately, we are faced with a minor dilemma. To properly present the following material to a non-expert requires quite some explanation. Thus we present an extremely superficial treatment. It will be assumed that the reader knows about the concepts in the field of quasi-spin and shell theory.

7.4.2 Tensorial Creation and Annihilation Operators

Previously in §5.4.4 *Creation and Annihilation Operators* we introduced the creation and annihilation operators. In what follows we will use creation and annihilation operators that have attached indices. These are needed when we consider "second-quantized" creation and annihilation operators, which we soon describe. First though, let us dispense with some of the formulation details of the problem in *Mathematica*. At this stage such details should be routine.

We define the notation for creation and annihilation operators as we did previously. The work here is just given a superficial treatment due to the size of the thesis.

```
In[1]:= Notation[a+ ⟺ CreationOp[a]]

In[2]:= Notation[a- ⟺ AnnihilationOp[a]]

In[3]:= AddInputAlias[a+ □, "cop"]

In[4]:= AddInputAlias[a- □, "aop"]
```

Here is an expression involving a collection of creation and annihilation operators.

```
In[5]:= a+ i · a- j · a- k · a+ l // FullForm

Out[5]//FullForm=
NonCommutativeTimes[Tensor[CreationOp[a], List[Low[i]]],
   Tensor[AnnihilationOp[a], List[Low[j]]],
   Tensor[AnnihilationOp[a], List[Low[k]]],
   Tensor[CreationOp[a], List[Low[l]]]]
```

Let us re-enter our normal ordering operation. `NormalOrder` is "multi-linear" as well as "expanding".

```
In[6]:= Assign[ GenericML_OpRules /. {
                GenericML_Op → NonCommutativeTimes,
                Expand → NormalOrder,
                Simplify → NormalOrder}]

In[7]:= Assign[Select[Values @ Expand, NonCommutativeTimes ∈?∞ rule &_rule] /.
            Expand → NormalOrder]

In[8]:= NormalOrder @ expr_ := DynamicBar @ expr

In[9]:= DeclareConstant[δ _ _]
```

Under fermionic shell theory we cannot have two electrons in the same state because of the Pauli exclusion principle [68, ,292]. The commutation relations for the second quantized operators are as follows.

$$\begin{aligned} a_i^\dagger\, a_j^\dagger &= -a_j^\dagger\, a_i^\dagger \\ a_i^\dagger\, a_j^\dagger &= -a_j^\dagger\, a_i^\dagger \\ a_i\, a_j^\dagger &= \delta_{ij} - a_j^\dagger\, a_i^\dagger \\ a_i^\dagger\, a_i^\dagger &= 0 \\ a_i\, a_i &= 0 \end{aligned} \tag{7.4.a}$$

By now, encoding such operations is easy.

```
In[10]:= NormalOrder ↗ (a⁻_i_ · a⁻_i_)_inp :=ᵃ 0

         NormalOrder ↗ (a⁻_i_ · a⁻_j_)_inp :=ᵃ (-a⁻_j · a⁻_i) /; j <ₜₑₓ i

         NormalOrder ↗ (a⁺_i_ · a⁺_j_)_inp :=ᵃ (-a⁺_j · a⁺_i) /; j <ₜₑₓ i

         NormalOrder ↗ (a⁻_i_ · a⁺_j_)_inp :=ᵃ (δ_i,j - a⁺_j · a⁻_i)

         NormalOrder ↗ (a⁺_i_ · a⁺_i_)_inp :=ᵃ 0

         NormalOrder ↗ (a⁻_i_ · a⁻_i_)_inp :=ᵃ 0
```

Here are some simple calculations involving the normal ordering of fermionic creation and annihilation operators.

```
In[16]:= a⁻_m · a⁺_i · a⁻_l · a⁺_j · a⁻_k

Out[16]= a⁻_m · a⁺_i · a⁻_l · a⁺_j · a⁻_k

In[17]:= a⁻_m · a⁺_i · a⁻_l // NormalOrder

Out[17]= a⁺_i · a⁻_l · a⁻_m + δ_m,i a⁻_l

In[18]:= a⁻_m · a⁺_i · a⁻_l · a⁺_j · a⁻_k // NormalOrder

Out[18]= a⁺_i · a⁺_j · a⁻_k · a⁻_l · a⁻_m + a⁺_i · a⁻_k · a⁻_m δ_l,j -
           a⁺_i · a⁻_k · a⁻_l δ_m,j + δ_m,i (a⁺_j · a⁻_k · a⁻_l + δ_l,j a⁻_k)
```

Results of this same sort can also be obtained by Wick's theorem [206]. Let us now discuss second quantization.

7.4.3 Introduction to Second Quantization

In the formalism of second quantization, the creation and annihilation operators do not just modify the eigenvalues of our eigenstates by creating or destroying a quantum of some label; instead they create or destroy whole particles. Consider a quantum harmonic oscillator. Creation and annihilation operators can act on this to create or destroy quanta of oscillation; however, just a single oscillator is involved. In the second-quantization formalism, we are in essence dealing with operators that create or destroy whole oscillators, *not* the quanta in the oscillator. This formalism is extensively used in quantum field theory [173, 186, 278]. However in field theories, the second-quantized creation and annihilation operators are actually quantum fields themselves. Also, there is a continuum of these operators ranging over the different momenta. In the "semi-classical" approach, we quantize the electromagnetic field but retain a non-relativistic formalism for our particles. This is quantum electrodynamics, or QED. In quantum field theory (QFT), the matter fields themselves become quantum operators. This gives rise to the term 'second quantization'. Such a formalism is necessary in an intrinsically many particle theory.

The second quantized operators used in this section are of a similar kind, yet different. We will use second quantization operators as applied to shell theory. These operators create and destroy particles but they are not fields. They create and destroy electrons in a given shell of an atom (or possibly nucleons in a nucleus). In this sense, we are not working in a field-theoretic formalism. Brink & Stachler were among the first to revive interest in applying the method of second quantization to shell theory. Amongst others, Judd[179], Lindgren & Morrison[206], and Weissbluth[334] provide details of this approach. In shell theory there are just a finite number of creation and annihilation operators for any given shell. These operators create or destroy specific electrons in the shell. For instance, we can write the state |1,0,1/2,-1/2⟩ as

$$\left|1_{\hat{L}},\ 0_{\hat{L_z}},\ (1/2)_{\hat{S}},\ (-1/2)_{\hat{S_z}}\right\rangle = \mathbf{a}^{+}_{\,L=1,\,m_l=0,\,S=1/2,\,m_s=-1/2}\cdot\left|0\right\rangle \qquad (7.4.b)$$

or more compactly,

$$\left|K\right\rangle = \mathbf{a}^{+}_{\,K}\left|0\right\rangle \qquad (7.4.c)$$

where K is a composite label containing all the individual labels. Within this framework, we can describe the quasi-spin operator.

The Cartesian components of the quasi-spin operators obey the same commutation relations as do the standard angular momentum relations, hence the name 'quasi-spin'. Historically, the quasi-spin operators were introduced in order to add particles to a shell in pairs. For instance, in atomic shell theory, we might add two electrons with opposite orbitial angular momenta and opposite spins to a shell, thus adding two electrons but adding no net angular momenta. The quasi-spin operators have application outside just theoretical considerations, for instance in Cooper pairs in superconductivity. Formally, up to some normalization coefficients which we soon explore, the spherical components of the quasi-spin operator can be stated as follows.

In[204]:=
$$\mathbf{Q[K]}_1 = (K\ \overline{K})_{2J}\ \frac{i}{\sqrt{2}}\ \mathbf{a}^+_K \cdot \mathbf{a}^+_{\overline{K}}$$
$$\mathbf{Q[K]}_0 = \frac{i}{2}\left(1 + \mathbf{a}^+_K \cdot \mathbf{a}^-_K - \mathbf{a}^+_{\overline{K}} \cdot \mathbf{a}^-_{\overline{K}}\right) \tag{7.4.d}$$
$$\mathbf{Q[K]}_{-1} = (\overline{K}\ K)^{\dagger}_{2J}\ \frac{i}{\sqrt{2}}\ \mathbf{a}^-_{\overline{K}} \cdot \mathbf{a}^-_K$$

It turns out that the results in Savage[284] appear to have problems in that it is not possible to find the correct normalization factors to make these objects obey the correct commutation relations. We explore these normalizations later in this section.

For the background on why the quasi-spin operator is a useful operator, what its significance is, and what the applications of such an operator are, one can consult Savage's thesis. Let us now progress onto describing the objects used in the definition of the quasi-spin operator and their semantics.

7.4.4 Conjugate States

In calculations involving angular momentum, a frequent concept that arises is that of a conjugate state. In our quasi-spin calculations, this is also the case. Thus, let us define the conjugate of one state with respect to another. The conjugate state to some state $|K\rangle$, call it $|\overline{K}\rangle$, is the unique state that couples to $|K\rangle$ to yield the zero state $|0\rangle$. That is, $|\overline{K}\rangle$ is the unique state such that

$$\langle 0 \mid K, \overline{K} \rangle\ |K\rangle \cdot |\overline{K}\rangle = |0\rangle \tag{7.4.e}$$

The Clebsch-Gordan coefficient $\langle 0 \mid K, \overline{K} \rangle$ is called a $2j$ symbol, and is unique. For an example involving LS coupling, here is a state and its conjugate

$$\overline{|\ell_{\hat{L}}, m_{\ell_{\hat{L}_z}}, S_{\hat{S}}, m_{s_{\hat{S}_z}}\rangle} = |\ell_{\hat{L}}, -m_{\ell_{\hat{L}_z}}, S_{\hat{S}}, -m_{s_{\hat{S}_z}}\rangle \tag{7.4.f}$$

Unfortunately, in symbolic computation we must again tackle the issue of context sensitive notations. In the context of ket reversal or conjugation, the overbar notation denotes time reversal. However, it shares the same appearance as complex conjugation. For instance, in the calculus of complex variables, a typical function might be

$$f(z) = \overline{z}\, z + \overline{z e^{i z}} \tag{7.4.g}$$

How then should we reconcile the dual use of the conjugation notation for both time reversal and complex conjugation? Even more importantly, if we have other labels not involved, do we time reverse those? Judd does not always do so, thus it appears that in general the conjugate state $|\overline{K}\rangle$ is not the same as the time reversed state.

$$|\overline{K}\rangle \neq \overline{|K\rangle} \tag{7.4.h}$$

This means we have three distinct concepts all using an overbar to denote them. All of these subtleties can lead to potential problems. In our case to disambiguate the situation we will use the notation $\overline{expr}$ to be a time reversed expression, $\overline{expr}$ to be a conjugate state, and $expr^*$ to be the complex conjugate of an expression. As usual we will denote the Hermitian conjugate of $expr$ by $expr^\dagger$.

```
In[19]:= Notation[OverBar[expr_] ⟺ StateConjugate[expr_]]
```

```
In[20]:= AddInputAlias[□, "sconj"]
```

Due to the nature of the conjugate state, we have the fact that double conjugation is equivalent to the original state. (However, this is not true for time reversal.)

```
In[21]:= OverBar[OverBar[K_]] = K;
```

Under the conventions of Savage[284] the expansions that are performed do not obey the Einstein summation convention. Thus we need to enter some rather non-standard rules. Let us add these rules to a new operator, say `QuasiNormalOrder`, so as not to pollute the functions of `NormalOrder`.

```
In[22]:= Assign[Values @ NormalOrder /. NormalOrder → QuasiNormalOrder]
```

Further to the above, the expansion of the commutator should also be performed by quasi normal ordering. We also make `QuasiNormalOrder` simplify the expressions it returns.

```
In[23]:= QuasiNormalOrder ↗ [x_, y_]_ := x · y - y · x
         QuasiNormalOrder @ any_ := DynamicBar @ Simplify @ any
```

Savage refers to all the quantum labels for a partial creation operator as K. The conjugate labels are referred to as L. The conjugate labels are those labels such that the K and L couple to the invariant of the group. In this case we have left the group open, but in practical situations an example might be SO(3) the group of 3 space rotations.

In fermionic systems, states cannot be self conjugate. We can thus enter the following simplifications

```
In[25]:= Tensor /: δ_{α_ -α_} := 0

         Tensor /: δ_{α_ α_} := 1

         QuasiNormalOrder ↗ Tensor /: δ_{K_ OverBar[K_]} := 0

         QuasiNormalOrder ↗ Tensor /: δ_{OverBar[K_] K_} := 0

         QuasiNormalOrder ↗ Tensor /: δ_{K_ K_} := 1
```

(This is not however true for bosonic angular momenta systems, for instance $\overline{|5_{\hat{L}}, 0_{\hat{L_z}}\rangle} = |5_{\hat{L}}, 0_{\hat{L_z}}\rangle$.)

Let us next introduce the 2j symbols and their conjugates.

7.4.5 2j Symbols and Phases

In the full definition of the quasi-spin operators we must include 2j phases and 2j symbols. In these calculations as many labels as possible have been suppressed in order to ease computational problems.

Let us create a notation and behavior for 2j symbols.

```
In[30]:= Notation[ (K_ L_)_2j ⟺ TwoJ[K_, L_]]
         Notation[ (K_ L_)†_2j ⟺ TwoJConjugate[K_, L_]]

In[32]:= TwoJ /: expr_TwoJ† := TwoJConjugate @@ expr
         TwoJConjugate /: expr_TwoJConjugate† := TwoJ @@ expr
```

The 2j symbols and their conjugates are constants.

```
In[34]:= DeclareConstant @ {_TwoJ, _TwoJConjugate}

In[35]:= (K_ L_)_2j := - (L K)_2j /; L <tex K
         (K_ L_)†_2j := - (L K)†_2j /; L <tex K

In[37]:= TwoJ /: (K_ L1_)†_2j (K_ L2_)_2j := δ_L1 L2
```

This rule generalizes to work over powers of TwoJ symbols.

```
In[38]:= QuasiNormalOrder ↗ ( ((K_ L1_)†_2j)^n_. (K_ L2_)^m_._2j )  :=
           ( With[ {p = Min[n, m]}, δ_L1 L2 ((K L1)†_2j)^(n-p) (K L2)^(m-p)_2j ] /; m > 0 ∧ n > 0 )
```

Here is an example of this behavior.

```
In[39]:= (L K)_2j^3 (L K)†_2j^2 // QuasiNormalOrder
Out[39]= - (K L)_2j
```

Technical Note: The powers have to be positive or else we can encounter expressions of the form $1/\infty$.

7.4.6 Quasi-Spin in Spherical Components

We now define the quasi-spin operator following Savage. It appears that this operator is not actually a quasi-spin operator, in that it does not satisfy the commutation relations required of it. Namely, (*i*) that it is a spherical tensor operator, and (*ii*) that it has the same commutation relations as that of normal angular momenta.

Here is the fundamental definition of the quasi-spin as given in Savage. (These are the spherical somponents of quasi-spin.)

```
In[40]:= Tensor /: Q[K]_1 := (K K̄)_{2 j} i/√2 a⁺_K · a⁺_K̄

         Tensor /: Q[K]_0 := i/2 (1 - a⁺_K · a⁻_K - a⁺_K̄ · a⁻_K̄)

         Tensor /: Q[K]_-1 := (K̄ K)†_{2 j} i/√2 a⁻_K̄ · a⁻_K
```

We can now verify the relations reported in Savage, namely relations (4.104) - (4.106).

```
In[43]:= [Q[K]_1, Q[K]_-1]_- == i Q[K]_0

Out[43]= [i a⁺_K · a⁺_K̄ (K K̄)_{2 j} / √2, -i a⁻_K̄ · a⁻_K (K K̄)†_{2 j} / √2]_- ==
           1/2 (-1 + a⁺_K · a⁻_K + a⁺_K̄ · a⁻_K̄)
```

Just for illustration's sake, we can expand the commutator in this equation with Expand since Expand still has the environment rules for commutators incorporated into it.

```
In[44]:= Expand @ %

Out[44]= -1/2 a⁻_K̄ · a⁻_K · a⁺_K · a⁺_K̄ δ_{K̄ K̄} + 1/2 a⁺_K · a⁺_K̄ · a⁻_K̄ · a⁻_K δ_{K̄ K̄} ==
           1/2 (-1 + a⁺_K · a⁻_K + a⁺_K̄ · a⁻_K̄)
```

Now by normal ordering the operators in this expression, and by simplifying the Kronecker delta functions, we can verify that the above equation is true.

```
In[45]:= %% // QuasiNormalOrder

Out[45]= True
```

In total, the spherical commutation relations of Savage are satisfied.

```
In[46]:= QuasiNormalOrder[{
           [Q[K]_1, Q[K]_-1]_- == i Q[K]_0,
           [Q[K]_-1, Q[K]_0]_- == -i Q[K]_-1,
           [Q[K]_1, Q[K]_0]_- == i Q[K]_1}]

Out[46]= {True, True, True}
```

Again, just to illustrate the computational complexity of attempting to do this by hand, here is the expansion of the above equation before normal ordering the creation and annihilation operators.

```
In[47]:= [Q[K]_1, Q[K]_0]_- == i Q[K]_1 // Expand
```

```
Out[47]= -1/(2 Sqrt[2]) ((a+[K] . a+[-K] - a+[K] . a+[-K] . a+[K] . a-[K] - a+[K] . a+[-K] . a+[-K] . a-[-K]) (K -K)[2,j]) -
  ((a+[K] . a+[-K] - a+[K] . a-[K] . a+[K] . a+[-K] - a+[-K] . a-[-K] . a+[K] . a+[-K]) (K -K)[2,j])/(2 Sqrt[2])

  == -(a+[K] . a+[-K] (K -K)[2,j])/Sqrt[2]
```

```
In[48]:= % // QuasiNormalOrder
Out[48]= True
```

7.4.7 Quasi-Spin in Cartesian Components

Since we are attempting to verify the results in Savage we will use the transformation matrix that he uses to transform between spherical and Cartesian components. This contra-standard Fano-Racah transformation matrix is i times the normal transformation matrix. Using this matrix, we obtain the Cartesian components of quasi-spin.

```
In[49]:= Tensor /: Q[K][x] = i/Sqrt[2] (Q[K][1] - Q[K][-1]) // QuasiNormalOrder // Expand
Out[49]= -1/2 a+[K] . a+[-K] (K -K)[2,j] + 1/2 a-[K] . a-[-K] (K -K)[2,j]†

In[50]:= Tensor /: Q[K][y] = 1/Sqrt[2] (Q[K][1] + Q[K][-1]) // QuasiNormalOrder // Expand
Out[50]= 1/2 i a+[K] . a+[-K] (K -K)[2,j] + 1/2 i a-[K] . a-[-K] (K -K)[2,j]†

In[51]:= Tensor /: Q[K][z] = -i (Q[K][0]) // Expand
Out[51]= 1/2 - 1/2 a+[K] . a-[K] - 1/2 a+[-K] . a-[-K]
```

The Cartesian components of quasi-spin should obey the normal commutation relations of angular momentum, that is, those given in §5.5.1 *Primitive Cartesian Angular Momenta*.

```
In[52]:= Q[K][x], Q[K][y]]- == -i Q[K][z] // QuasiNormalOrder
Out[52]= True

In[53]:= Q[K][x], Q[K][z]]- == i Q[K][y] // QuasiNormalOrder
Out[53]= True

In[54]:= Q[K][y], Q[K][z]]- == -i Q[K][x] // QuasiNormalOrder
```

Out[54]= True

Collectively, this shows that $\left[\mathbf{Q[K]}_i, \mathbf{Q[K]}_j\right]_- == -i\,\epsilon_{ijk}\,\mathbf{Q[K]}_k$. This is the opposite of what we need since the normal commutation relations of angular momentum are $\left[\mathbf{J}_i, \mathbf{J}_j\right]_- == i\,\epsilon_{ijk}\,\mathbf{J}_k$. Thus, this current formulation of quasi-spin is flawed. However, we will persevere with it and confirm the relations shown in Savage[284] before attempting to fix these normalization problems.

7.4.8 Particle-Hole Conjugation

We now skim the treatment of particle-hole conjugation. Our first step is to symbolize the operators to be used.

```
In[55]:= Symbolize[C_ℒ, SymbolizeRootName → "ParticleHoleConjugation"]
         Symbolize[C_ℒ^-1, SymbolizeRootName → "InverseParticleHoleConjugation"]
```

The following relations give the commutation relations of the creation and annihilation operators with the linear particle-hole conjugation operator, $C_{\mathcal{L}}$ and its inverse $C_{\mathcal{L}}^{-1}$. This is the operator that turns particles into holes and vice-versa.

```
In[57]:= QuasiNormalOrder ↗ (C_ℒ · a⁻_K_)_hp :=^a (K K̄)_2j a⁺_K̄ · C_ℒ

         QuasiNormalOrder ↗ (C_ℒ · a⁻_K_)_hp :=^a (K K̄)_2j a⁺_K̄ · C_ℒ

         QuasiNormalOrder ↗ (C_ℒ · a⁺_K_)_hp :=^a (K K̄)†_2j a⁻_K̄ · C_ℒ

         QuasiNormalOrder ↗ (a⁻_K_ · C_ℒ^-1)_hp :=^a (K K̄)_2j C_ℒ^-1 · a⁺_K̄

         QuasiNormalOrder ↗ (a⁺_K_ · C_ℒ^-1)_hp :=^a (K K̄)†_2j C_ℒ^-1 · a⁻_K̄
```

Finally, the operator $C_{\mathcal{L}}$ is the inverse of $C_{\mathcal{L}}^{-1}$.

```
In[62]:= QuasiNormalOrder ↗ (C_ℒ · C_ℒ^-1)_hp :=^a 1
```

The Cartesian components of the quasi-spin operators commute with the particle-hole operators.

```
In[63]:= (C_ℒ · Q[K]_x · C_ℒ^-1) == -(Q[K]_x)
```

$$\text{Out[63]= } C_{\mathcal{L}} \cdot \left(-\frac{1}{2}\,\mathbf{a}^+_{K} \cdot \mathbf{a}^+_{\bar{K}}\,(K\,\bar{K})_{2j} + \frac{1}{2}\,\mathbf{a}^-_{K} \cdot \mathbf{a}^-_{\bar{K}}\,(K\,\bar{K})^\dagger_{2j}\right) \cdot C_{\mathcal{L}}^{-1} == \frac{1}{2}\,\mathbf{a}^+_{K} \cdot \mathbf{a}^+_{\bar{K}}\,(K\,\bar{K})_{2j} - \frac{1}{2}\,\mathbf{a}^-_{K} \cdot \mathbf{a}^-_{\bar{K}}\,(K\,\bar{K})^\dagger_{2j}$$

```
In[64]:= Expand @ %
```

```
Out[64]= -1/2 C_L · a+_K · a+_K̄ · C_L^-1 (K K̄)_2j + 1/2 C_L · a-_K · a-_K̄ · C_L^-1 (K K̄)†_2j ==
         1/2 a+_K · a+_K̄ (K K̄)_2j - 1/2 a-_K · a-_K̄ (K K̄)†_2j
```

```
In[65]:= QuasiNormalOrder @ %
Out[65]= True
```

```
In[66]:= (C_L · Q[K]_y · C_L^-1) == (Q[K]_y) // QuasiNormalOrder
Out[66]= True
```

```
In[67]:= (C_L · Q[K]_z · C_L^-1) == -Q[K]_z // QuasiNormalOrder
Out[67]= True
```

However the spherical components are conjugated by the particle-hole operator.

```
In[68]:= (C_L · Q[K]_1 · C_L^-1) == Q[K]_-1 // QuasiNormalOrder
Out[68]= True
```

```
In[69]:= (C_L · Q[K]_-1 · C_L^-1) == Q[K]_1 // QuasiNormalOrder
Out[69]= True
```

```
In[70]:= (C_L · Q[K]_0 · C_L^-1) == -Q[K]_0 // QuasiNormalOrder
Out[70]= True
```

7.4.9 Time Reversal Commutation Relations

Time reversal acts on states by reversing the direction of motion or time, and also by complex conjugation.

```
In[71]:= Notation[expr_ ⟺ TimeReversal[expr_]]
```

The action of time reversal distributes over the normal arithmetic operators.

```
In[72]:= Act ↗ product_NonCommutativeTimes := # & /@ product
         Act ↗ product_Times := # & /@ product
         Act ↗ sum_Plus := # & /@ sum
```

For pure constants, all that time reversal does is to complex conjugate them.

```
In[75]:= c_?ConstQ := c†
```

Here is a typical example involving the action of time reversal.

```
In[76]:= Act @ a (b + c)
```

```
Out[76]= a̅ (b̅ + c̅)

In[77]:= Act ↗ a⁺_K_ := (K K̅)_2j a⁺_K̅

         Act ↗ a⁻_K_ := (K K̅)†_2j a⁻_K̅

In[79]:= Act @ any_ := DynamicBar @ any
```

Here is how the creation and annihilation operators respond to time reversal.

```
In[80]:= a⁺_K

Out[80]= a⁺_K

In[81]:= Act @ %

Out[81]= -δ_K̅ K̅ a⁺_K

In[82]:= QuasiNormalOrder @ %

Out[82]= -a⁺_K

In[83]:= a⁻_K // Act // QuasiNormalOrder

Out[83]= -a⁻_K
```

The particle-hole conjugation operator is time even.

```
In[84]:= C_ℒ = C_ℒ;

In[85]:= C_ℒ^-1 = C_ℒ^-1;
```

7.4.10 The Inter-Relations of Time Reversal

Here is how time reversal acts on the Cartesian components of quasi-spin

```
In[86]:= (Q[K]_x == Q[K]_x) // Act // QuasiNormalOrder

Out[86]= True

In[87]:= (Q[K]_y == -Q[K]_y) // Act // QuasiNormalOrder

Out[87]= True

In[88]:= (Q[K]_z == Q[K]_z) // Act // QuasiNormalOrder
```

```
Out[88]= True
```

Here is how time reversal interacts with particle-hole conjugation

```
In[89]:= $\overline{\left(C_{\mathcal{L}} \cdot \mathbf{Q}[\mathbf{K}]_{x} \cdot C_{\mathcal{L}}^{-1}\right)} == -\left(\mathbf{Q}[\mathbf{K}]_{x}\right)$
```

$$\text{Out[89]}= \overline{C_{\mathcal{L}} \cdot \left(-\frac{1}{2}\,\mathbf{a}^{+}_{K} \cdot \mathbf{a}^{+}_{\overline{K}}\,(K\,\overline{K})_{2\,J} + \frac{1}{2}\,\mathbf{a}^{-}_{K} \cdot \mathbf{a}^{-}_{\overline{K}}\,(K\,\overline{K})^{\dagger}_{2\,J}\right) \cdot C_{\mathcal{L}}^{-1}} == \frac{1}{2}\,\mathbf{a}^{+}_{K} \cdot \mathbf{a}^{+}_{\overline{K}}\,(K\,\overline{K})_{2\,J} - \frac{1}{2}\,\mathbf{a}^{-}_{K} \cdot \mathbf{a}^{-}_{\overline{K}}\,(K\,\overline{K})^{\dagger}_{2\,J}$$

```
In[90]:= % // Act // QuasiNormalOrder
Out[90]= True
```

```
In[91]:= $\overline{\left(C_{\mathcal{L}} \cdot \mathbf{Q}[\mathbf{K}]_{y} \cdot C_{\mathcal{L}}^{-1}\right)} == -\left(\mathbf{Q}[\mathbf{K}]_{y}\right)$
```

$$\text{Out[91]}= \overline{C_{\mathcal{L}} \cdot \left(\frac{1}{2}\,i\,\mathbf{a}^{+}_{K} \cdot \mathbf{a}^{+}_{\overline{K}}\,(K\,\overline{K})_{2\,J} + \frac{1}{2}\,i\,\mathbf{a}^{-}_{K} \cdot \mathbf{a}^{-}_{\overline{K}}\,(K\,\overline{K})^{\dagger}_{2\,J}\right) \cdot C_{\mathcal{L}}^{-1}} == -\frac{1}{2}\,i\,\mathbf{a}^{+}_{K} \cdot \mathbf{a}^{+}_{\overline{K}}\,(K\,\overline{K})_{2\,J} - \frac{1}{2}\,i\,\mathbf{a}^{-}_{K} \cdot \mathbf{a}^{-}_{\overline{K}}\,(K\,\overline{K})^{\dagger}_{2\,J}$$

```
In[92]:= % // Act // QuasiNormalOrder
Out[92]= True
```

```
In[93]:= $\overline{\left(C_{\mathcal{L}} \cdot \mathbf{Q}[\mathbf{K}]_{z} \cdot C_{\mathcal{L}}^{-1} \text{ // Expand}\right)} == -\left(\mathbf{Q}[\mathbf{K}]_{z}\right)$
```

$$\text{Out[93]}= \frac{1}{2}\,C_{\mathcal{L}} \cdot C_{\mathcal{L}}^{-1} - \frac{1}{2}\,C_{\mathcal{L}} \cdot \mathbf{a}^{+}_{K} \cdot \mathbf{a}^{-}_{K} \cdot C_{\mathcal{L}}^{-1} - \frac{1}{2}\,C_{\mathcal{L}} \cdot \mathbf{a}^{+}_{\overline{K}} \cdot \mathbf{a}^{-}_{\overline{K}} \cdot C_{\mathcal{L}}^{-1} == -\frac{1}{2} + \frac{1}{2}\,\mathbf{a}^{+}_{K} \cdot \mathbf{a}^{-}_{K} + \frac{1}{2}\,\mathbf{a}^{+}_{\overline{K}} \cdot \mathbf{a}^{-}_{\overline{K}}$$

```
In[94]:= % // Act // QuasiNormalOrder
Out[94]= True
```

7.4.11 Quasi-Spinors

We now superficially introduce the concept of quasi-spinors. The creation and the symmetrized annihilation operators transform irreducibly in quasi-spin space. That is, they are spherical tensor operators in quasi-spin space. Formally, the rank 1/2 operators, $\mathbf{a}^{+}_{K}$ and $\tilde{\mathbf{a}}^{-}_{K}$ transform irreducibly, where $\tilde{\mathbf{a}}^{-}_{K}$ is given by

```
In[95]:= $\tilde{\mathbf{a}}^{-}_{K\_} := (K\,\overline{K})^{\dagger}_{2\,J}\,\mathbf{a}^{-}_{\overline{K}}$
```

To show that these operators satisfy the properties of spherical tensor operators in quasi-spin space we must verify the following relations (exactly like we previously did in §5.5 *Example: Angular Momentum*).

In[96]:= $\left[\mathbf{Q[K]}_0, \mathbf{a^+}_K\right]_- == \frac{1}{2}\,\mathbf{a^+}_K$ // QuasiNormalOrder

Out[96]= $\left(-\frac{1}{2} - \frac{i}{2}\right)\mathbf{a^+}_K == 0$

In[97]:= $\left[\mathbf{Q[K]}_{-1}, \mathbf{a^+}_K\right]_- == \frac{1}{\sqrt{2}}\,\tilde{\mathbf{a}}^-_K$ // QuasiNormalOrder

Out[97]= $-\frac{(1+i)\,\mathbf{a^-}_{\overline{K}}\,(K\,\overline{K})^{\dagger}_{2j}}{\sqrt{2}} == 0$

Unfortunately the formulation of Savage also does not work here.

7.4.12 Corrected Components of Quasi-Spin

To find the correct components of quasi-spin which satisfy the Cartesian and spherical commutation relations we simply insert unsolved coefficients into the quasi-spin operator and then back solve for these coefficients from the correct relations. We omit this step. Here are the final correct spherical components of quasi-spin.

In[98]:= Tensor /: $\mathbf{Q[K]}_1 := \frac{-1}{\sqrt{2}}\,(K\,\overline{K})_{2j}\,\mathbf{a^+}_K \cdot \mathbf{a^+}_{\overline{K}}$

Tensor /: $\mathbf{Q[K]}_0 := \frac{-1}{2}\left(1 - \mathbf{a^+}_K \cdot \mathbf{a^-}_K - \mathbf{a^+}_{\overline{K}} \cdot \mathbf{a^-}_{\overline{K}}\right)$

Tensor /: $\mathbf{Q[K]}_{-1} := \frac{-1}{\sqrt{2}}\,(\overline{K}\,K)^{\dagger}_{2j}\,\mathbf{a^-}_{\overline{K}} \cdot \mathbf{a^-}_K$

With these relations the quasi-spin operators are spherical tensor operators in quasi-spin space.

In[101]:= $\left[\mathbf{Q[K]}_0, \mathbf{a^+}_K\right]_- == \frac{1}{2}\,\mathbf{a^+}_K$ // QuasiNormalOrder

Out[101]= True

In[102]:= $\left[\mathbf{Q[K]}_{-1}, \mathbf{a^+}_K\right]_- == \frac{1}{\sqrt{2}}\,\tilde{\mathbf{a}}^-_K$ // QuasiNormalOrder

Out[102]= True

In this formulation it is interesting to observe the squared components of quasi-spin.

In[103]:= $\mathbf{Q[K]}_x \cdot \mathbf{Q[K]}_x + \mathbf{Q[K]}_y \cdot \mathbf{Q[K]}_y + \mathbf{Q[K]}_z \cdot \mathbf{Q[K]}_z$

```
In[103]:= (1/2 - 1/2 a+[K] . a-[K] - 1/2 a+[K̄] . a-[K̄]) . (1/2 - 1/2 a+[K] . a-[K] - 1/2 a+[K̄] . a-[K̄]) +
          (1/2 i a+[K] . a+[K̄] (K K̄)[2,j] + 1/2 i a-[K] . a-[K̄] (K K̄)†[2,j]) .
            (1/2 i a+[K] . a+[K̄] (K K̄)[2,j] + 1/2 i a-[K] . a-[K̄] (K K̄)†[2,j]) +
          (-1/2 a+[K] . a+[K̄] (K K̄)[2,j] + 1/2 a-[K] . a-[K̄] (K K̄)†[2,j]) .
            (-1/2 a+[K] . a+[K̄] (K K̄)[2,j] + 1/2 a-[K] . a-[K̄] (K K̄)†[2,j])

In[104]:= Expand @ %

Out[104]= 1/4 - 1/2 a+[K] . a-[K] - 1/2 a+[K̄] . a-[K̄] +
          1/4 a+[K] . a-[K] . a+[K] . a-[K] + 1/4 a+[K] . a-[K] . a+[K̄] . a-[K̄] +
          1/4 a+[K̄] . a-[K̄] . a+[K] . a-[K] + 1/4 a+[K̄] . a-[K̄] . a+[K̄] . a-[K̄] -
          1/2 a-[K] . a-[K̄] . a+[K] . a+[K̄] δ[K̄,K̄] - 1/2 a+[K] . a+[K̄] . a-[K] . a-[K̄] δ[K̄,K̄]

In[105]:= Expand @ QuasiNormalOrder @ %

Out[105]= 3/4 - 3/4 a+[K] . a-[K] - 3/4 a+[K̄] . a-[K̄] - 3/2 a+[K] . a+[K̄] . a-[K] . a-[K̄]
```

Thus, we can see that Q^2 is equivalent to something like $\frac{3}{4}\left(1 - n(K) - n(\overline{K}) + 2\,n(K)\cdot n(\overline{K})\right)$.

Finally, let us note that by using the standard transformation between the Cartesian components and the spherical components we obtain the correct commutation relations of Cartesian angular momenta.

```
In[106]:= Tensor /: Q[K][x] = -1/√2 (Q[K][1] - Q[K][-1]) // QuasiNormalOrder // Expand

Out[106]= 1/2 a+[K] . a+[K̄] (K K̄)[2,j] - 1/2 a-[K] . a-[K̄] (K K̄)†[2,j]

In[107]:= Tensor /: Q[K][y] = i/√2 (Q[K][1] + Q[K][-1]) // QuasiNormalOrder // Expand

Out[107]= -1/2 i a+[K] . a+[K̄] (K K̄)[2,j] - 1/2 i a-[K] . a-[K̄] (K K̄)†[2,j]

In[108]:= Tensor /: Q[K][z] = (Q[K][0]) // Expand

Out[108]= -1/2 + 1/2 a+[K] . a-[K] + 1/2 a+[K̄] . a-[K̄]
```

Here are the new commutation relations for the Cartesian components of quasi-spin.

```
In[109]:= [Q[K][x], Q[K][y]]- == i Q[K][z] // QuasiNormalOrder
```

```
Out[109]= True

In[110]:= [Q[K]_x, Q[K]_z]_- == -i Q[K]_y // QuasiNormalOrder

Out[110]= True

In[111]:= [Q[K]_y, Q[K]_z]_- == i Q[K]_x // QuasiNormalOrder

Out[111]= True
```

Our corrected quasi-spin operator satisfies the correct commutation relations for cartesian angular momentum, namely $\left[\mathbf{Q[K]}_i, \mathbf{Q[K]}_j\right]_- == i\,\epsilon_{ijk}\,\mathbf{Q[K]}_k$. Moreover, it can easily be show that the other commutation relations are suitably satisfied by our new formulation of quasi-spin.

There are many other relations we could show with these operations. Again for the background and concepts behind these operations consult Savage[284]. However, the above examples should provide a "taste" of what is possible. The important realization that comes from the calculations presented in this subsection is that once the foundational structures are in place it is now much easier to create our computations. We can now create computations with an ease that was unparalleled before. The foundations of these calculations are exactly the same as those previously given in the section §5 *Prototypical Structures and Quantum Mechanics*.

Chapter 8
Conclusions

8.1 Summary

This thesis presents original contributions to the capabilities of symbolic computation systems in general, and *Mathematica* in particular. The developments are interdisciplinary in nature, but the applications given in this thesis are to physics.

Since almost all of the constituent chapters have a detailed introduction and conclusion, we forego a detailed summary here. Instead, the following is an extremely brief summary of the chapters that comprise this thesis.

Chapters 2 and 3 presented a thorough description of the features of the *Notation* package. These chapters described what can be considered a major advance in computer algebra interface systems, at least to the extent that such overall interfaces were not available before this work. Of course the system is built on top of *Mathematica*'s already extremely advanced system (at least at the time of this thesis), yet our advances should not be underestimated. The setting and representations in which we choose to carry out our calculations can now be tailored to the specific notations of a given field. The functions in the *Notation* package automatically set up parsing and formatting rules for the specified notations, including the correct parenthesizing, styling, and precedences.

Even with the Notation package, it is a non-trivial task to introduce properly *functioning* notations for some standard notions common in physics. Chapter 2 culminated in an exposition of the development and details leading up to a complete and functioning implementation of Dirac's bra-ket notation. In Chapter 3 a corresponding development was presented for tensorial notations. To the best of the author's knowledge, neither of these tasks has been achieved before. Also, the latter should be of immediate significance, considering the large number of recognized tensor packages available, none of which have a properly functioning tensorial notation.

Chapter 4 presented our language extensions. These extensions, embodied in the *Assign* package, center around a prototypical inheritance system that provides a context sensitive non-local rewrite rule paradigm. From a computer science viewpoint, our developments are radical since they enact a model whereby the "code" of a program changes dynamically as evaluation proceeds.

Various applications to physics, of the language modifications and notational advancements, were given in Chapter 5. In that chapter many standard topics from quantum mechanics were

presented from a perspective in keeping with the theoretical developments of the foundational Chapters 2 through 4. The style of the calculations presented in that chapter are truly elegant in comparison to the attempts of many others.

Tensors are ubiquitous in modern physics, from general relativity to modern field theories. In symbolic computation and indeed in the wider field, one often needs to simplify complicated tensorial expressions, consisting of possibly thousands of terms. Chapter 6 presented an algorithm to canonicalize tensorial expressions. It in essence removed the problem of dummy index relabeling by an innovative relabeling scheme. After presenting a basic canonicalization routine based on this innovation, the chapter proceeded on to consider other original developments needed to yield the optimized canonicalization routine. This optimized routine was then further extended, to handle full linear symmetries and other equational symmetries. Indeed, the algorithm presented handles mixed index classes, so we can naturally obtain 1+3 splits. Moreover, it also handles all the subtleties necessary when canonicalizing expressions with partial derivatives.

In the final chapter, we presented some substantial applications of our developments to selected problems in physics. We presented the operations for tensor calculus. The expressive power of our extensions was fully illustrated. It is evident that our language models and our canonicalization routines have combined to yield a system in which calculations are "natural" and readily extensible. Indeed, the author has found that most of the calculations are recognizable to physicists who have had little exposure to symbolic systems, if any at all.

The appendices of this thesis variously contain: some common notations, a discussion of attributes and their interaction with our models, permutation group conventions, a system for machine verifying our conjectures up to a certain stage, a system for examining the direct reduction method in the canonicalization algorithm, and the code for the heart of the canonicalization algorithm.

8.2 Conclusions

Since all of the major chapters include conclusions, we will end this work with only the briefest of conclusions.

We have chosen to make our early treatment simple enough that many readers would understand all of the presented material. For contrast, in the final chapter, we presented some physics computations "in the raw". It is hard finding the exact balance of readability, but at the same time exposing the powers of ones system. Also the varying backgrounds of readers from the disciplines of physics, computer science, and mathematics, mean that different parts of different sections will be challenging to different readers. We attempted to strike the right balance with these considerations in mind.

There are several directions in which we could pursue further research based on the work in this thesis. The most direct applications of the developments herein are to physics. We could treat

an extremely large class of problems, since the approach taken here is from a general perspective. Certainly, the applications given in Chapter 5 and 7 could be extended in myriad ways. Of course, certain specific problems can be much more efficiently "hard coded". However, in many problems in symbolic computation, solution times turn out to be exponential in "problem size". So even hard coding a problem will not always yield that much of an improvement. Thus in a very real sense, it matters more that we can state our problems in a flexible and extensible way.

It should be pointed out that although it is remotely possible that the canonicalization algorithm may not yield the most canonical answer, it should *never* give a wrong answer. The algorithm appears to be orders of magnitude faster than other comparable algorithms which are "full" algorithms. Finally, we once again reiterate that it would seem that direct canonicalization has vast potential for improving the speed of our algorithm.

The physics applications that have been experimented with outside those directly documented in this thesis involve non-commutative canonicalization, Dirac algebras, general relativity, and field theory. We can treat *certain* symmetries in non-commutative products with our canonicalization algorithm by removing degeneracy symmetries or introducing new symmetries. The basis whereby one should proceed to tackle these issues should be clear.

Even though the computer science aspects of our extensions are extremely interesting, as commented previously, most modern computer languages are strongly typed and have leanings towards "formality". This, however, does not necessarily preclude our developments from being integrated into a highly theoretical or rigorous model. In some sense, we could describe the developments in this thesis as "experimental language design / computer science" as applied to physics. Certainly as previously stated, the rewrite rule paradigm employed by *Mathematica* is quite advanced. Our inheritance paradigm builds on *Mathematica*'s foundation and extends it in an extremely elegant but non-trivial way.

It is extremely important in computer algebra for our computations to be recognizable. This is not just for the use of others, but also for the ability to create and manage the calculations oneself. This can be achieved, as we have demonstrated, through both an appropriate language model and a sufficiently general interface, such as that provided by the *Notation* package integrated with the *Mathematica* front end.

We close with one final reiterated comment. At the end of Chapter 3 we mentioned that the expansion of the Christoffel symbol in an alternative computer algebra system was written as follows.

```
withmetric =
   {Gam[i_, k_, l_] :→
         Module[{m},
            DefES[1 / 2 gc[i, m] * (PD[g[m, k], x[l]] +
                        PD[g[m, l], x[k]] - PD[g[k, l], x[m]]),
               {m}, ESRange → $ESDimension]]};
```

Truly, this could only be comprehensible to a select few. Alternatively, here is how this expression can now be written in *Mathematica*, subsequent to the developments of this thesis.

$$\text{ResolveΓ} \nearrow \Gamma^{i\_}{}_{k\_l\_} := \frac{1}{2} g^{im} \left(g_{km,l} + g_{ml,k} - g_{kl,m} \right)$$

This is but a small illustration of our overall goal. I believe that in the end, we, the practitioners, will only really be satisfied with our symbolic computation systems once the programs we create and the formulas that we write are essentially one and the same.

It is my belief that the work of this thesis breaks new ground in this and other regards. Hopefully, it will be one of the steps in the staircase upon which symbolic computation systems must travel in order to reach Zielberger's *marvelous computer-mathematics revolution of the late 21st century*.

Appendix A
Notational Issues

A.1 Common Notations

Throughout the body of the thesis it is convenient to use some common notations which have broad applicability. These notations are in addition to the specific notations used in individual sections, for instance, say the notations in §3.4 *Tensorial Notation*, §6 *An Algorithm for Tensor Simplification*, and §2.7 *Example: Bra-Ket Notation*.

The following notations are defined in the package `CommonNotations`. The package `CommonNotations`, where possible, creates these notations in a context independent way as described in §3.2.3 *Tags in Tag Boxes*.

```
InfixNotation[+⊕, Join]
Notation[patt_hold ⟺ HoldPattern[patt_]]
Notation[patt_hc ⟺ HoldComplete[patt_]]
Notation[expr_u ⟺ Unevaluated[expr_]]
Notation[body_&λ_ ⟺ Function[{λ__}, body_]]
Notation[body_&λ_attr_ ⟺ Function[{λ__}, body_, {attr__}]]
AddInputAlias["function" → □&□]
InfixNotation[≢, UnsameQ]
InfixNotation[≡, SameQ]
Notation[elem_ ∈? expr_ ⟺ MemberQ[expr_, elem_]]
Notation[elem_ ∈?level_ expr_ ⟺ MemberQ[expr_, elem_, level_]]
Notation[elem_ ∉? expr_ ⟺ FreeQ[expr_, elem_, 1]]
Notation[elem_ ∉?level_ expr_ ⟺ FreeQ[expr_, elem_, level_]]
(Notation[!f testFunction_ ⟺ NotFunction[testFunction_]]
!)f test_ = ! test[#] &;
silentEvaluate @ Notation[patt_?¬test_ ⟹ patt_?(!f test_)]
silentEvaluate @ Notation[patt_?! test_ ⟺ patt_?(!f test_)]
Notation[ℓ_len ⟺ Length[ℓ_]]
Notation[a_ ≼lex b_ ⟺ OrderedQ[{a_, b_}]]
Notation[a_ ⋠lex b_ ⟹ Not[OrderedQ[{a_, b_}]]]
Notation[a_ ≺lex b_ ⟹ Not[OrderedQ[{b_, a_}]]]
InfixNotation[\, Complement]
Notation[a_ \⧵ b_ ⟺ UnorderedComplement[a_, b_]]
```

```
list_List \ₓ out_List := DeleteCases[list, Alternatives @@ out]
Notation[l_ ⁱʰᵖ→ r_ ⟺ l_ₕₚ → r_]
Notation[l_ ⁱʰᵖ⇢ r_ ⟺ l_ₕₚ ⇢ r_]
```

Appendix B
Language Modifications

B.1 Attributes, Pattern Matching, and Associativity

Included in the *Mathematica* language are attributes. Attributes allow designers to specify certain facts about functions, for instance, that they are constant (`Constant`), or protected (`Protected`), etc. This subsection focuses on the attributes which affect pattern matching, specifically, the attributes `Flat`, `OneIdentity`, and `Orderless`.

The general purpose of the `Flat` attribute is to state that a function is "flat", for instance, that $f[a, f[b, c]]$ is the same as $f[a, b, c]$. More documentation on attributes can be obtained from the *Mathematica* book [343] or from [222, 312, 325, 123]. Let us give an example where we would like to introduce a flat function. In §5 *Prototypical Structures and Quantum Mechanics* we introduced creation and annihilation operators. For fermionic operators, the commutation relations of the operators satisfy the following. (We use traditional notation in the following.)

$$\begin{aligned} a_i^\dagger\, a_j^\dagger &= -a_j^\dagger\, a_i^\dagger \\ a_i^\dagger\, a_j^\dagger &= -a_j^\dagger\, a_i^\dagger \\ a_i\, a_j^\dagger &= \delta_{ij} - a_j^\dagger\, a_i^\dagger \end{aligned} \tag{B.1.a}$$

Say we entered a rule trying to implement relations (B.1.a). For example, consider the following.

```
a⁻_i_ · a⁺_j_ := (δ_i_j - a⁺_j · a⁻_i) /; j <tex i
```
(B.1.b)

We will need the rule to act correctly with expressions like `a⁻_m · a⁺_i · a⁻_l · a⁺_j · a⁻_k`. Yet, the hypothetical rule (B.1.b) would not match any parts of the example expression since the left hand side of the rule has only two arguments while the expression has five. We need pattern matching to "look inside" the non-commutative head and pattern match by regrouping objects. For instance, regrouping `a⁻_m · a⁺_i · a⁻_l · a⁺_j · a⁻_k` as `a⁻_m · a⁺_i · (a⁻_l · a⁺_j) · a⁻_k` would make it possible to apply (B.1.b). Thus, it would be desirable to use an attribute like `Flat`.

Technical Note: We could easily load up the various notations to demonstrate the validity of the paragraph above, but after the material in §3 *Foundations of Notation*, the argument should be self-evident.

Unfortunately, there is a problem with using the `Flat` attribute. To illustrate this problem, note the following behavior.

```
In[1]:= Plus[a]
Out[1]= a

In[2]:= Times[a]
Out[2]= a
```

Both `Plus` and `Times`, two fundamental and extremely common operations in *Mathematica*, are equivalent to the identity when acting on a single element. We would usually like the same behavior to be true of our operators, for instance, our non-commutative times operation. Yet, if we set the attributes of our operator to `Flat` and `OneIdentity`, this will lead to infinite recursion. Let us illustrate this for a hypothetical operator `myPlus`. We will attempt to make `myPlus` mimic the standard `Plus`.

```
In[3]:= ClearAll[myPlus]

In[4]:= SetAttributes[myPlus, {Flat}]

In[5]:= myPlus[any_] := any
```

Yet, unfortunately with the addition of this simple rule, any subsequent expressions involving `myPlus` fail through infinite recursion.

```
In[6]:= myPlus[x, y]
      $IterationLimit::itlim : Iteration limit of 4096 exceeded.
Out[6]= Hold[myPlus[x, y]]
```

So it appears that we cannot set the `Flat` attribute for a function if we need to have that function act like the identity on single elements. This is a huge restriction since most normal additivity or multiplicative operators or operations are of this kind. What to do? Typically the approach taken is to just define one's own rules to mimic the `Flat` and associative nature of our operation. This is the approach taken in this thesis. In particular, see §5.2.2 *Generic Linearity, Associativity, and Flatness.*

There is a way around this fiasco, but one should treat the following approach with trepidation. If a rule is entered before an attribute is set, then the rule is unaffected by the attribute. For instance:

```
In[7]:= myTimes[any_] := any

In[8]:= SetAttributes[myTimes, {Flat, OneIdentity}]

In[9]:= myTimes[x, y]
Out[9]= myTimes[x, y]
```

```
In[10]:= myTimes[x]

Out[10]= x
```

This *temporal attribute behavior* appears to span at least several known versions of *Mathematica*. Specifically, this behavior is true for *Mathematica* versions 2.x, 3.x, and 4.x. In addition, this behavior has also been confirmed to be true for an early alpha of version 5.X.X [private communication with Robby Villegas].

This raises several issues. First, is it a bug that flat functions infinitely recurse for the simple rules above? The answer is strictly no, since `Flat` is working according to the documented specification, yet intuitively this leaves much to be desired. Second, given that there exists a method to circumvent this bug / feature, should we adapt the rules returned by `Values` to turn off the bug / feature? For instance, it would not be terribly difficult to have `Values` return a modified list of rules, such that any of the rules which we can ascertain would lead to problems, would be surrounded by statements which would turn the offending attribute(s) off, and then back on.

```
Values @ f ⟶ {
      …,
          AttributeIsSetQ[f, OneIdentity]  → False,
          AttributeIsSetQ[f, Flat]         → False,
          f[any_]                          → any,                    (B.1.c)
          AttributeIsSetQ[f, OneIdentity]  → True,
          AttributeIsSetQ[f, Flat]         → True,
      …}
```

Currently the *Assign* package does not do this. There are several reasons why it does not. Firstly, code like (B.1.c) might be viewed as working around a bug. Secondly, it might be difficult to ascertain what other rules besides the single identity rule could be affected. Thirdly, such approaches using `Flat` are dangerous.

Even though the *Assign* package does not explicitly support temporal attribute assignment code like (B.1.c), by creating rule sets such as that in (B.1.c) manually, one can still obtain the overall temporal attribute behavior. However there are other possible reasons not to use the attributes `Flat` and `OneIdentity`. These concern efficiency. Due to space considerations the following result will only be quoted. If one uses operators with the `Flat` attribute, then due to the attempts of `Flat` to "rearrange expressions in order to accommodate pattern matching", pattern matching can become markedly slower with more complicated expressions. This can easily be verified by (*i*) entering some rules for factoring out constants, and then say creating a non-commutative product of maybe 100 terms and timing the expansion of the product; and (*ii*) comparing the factoring of constants in the exact same expression when using rules to implement flatness — see §5.3.5 *Efficient Matching*. Typically the latter implementation will be orders of magnitude faster on larger expressions.

It is likely that the drop in efficiency due to using `Flat` is primarily due to the pattern matching trying so many different combinations in order to match the pattern.

Appendix C Canonicalization Topics

C.1 Permutation and Group Action Conventions in *Mathematica*

In the subsections of §6.3 *Permutations and Configurations* and §6.4 *Labeling, Relabeling, and Group Actions* the group action $*_c$ was developed in parallel with the group operation $\cdot$. This appendix enters into more detail on how the design decisions were entered into concerning how to define the various operations. First we need to describe the concepts of an operator acting "from right to left" and "from left to right".

In the following we will refer to binary operations as "multiplication", even though, strictly speaking, some particular binary operation might not be multiplication but rather a group operation, or function application, or otherwise. For instance, in quantum mechanics it is ubiquitous to promote the position and momentum operators $\hat{P}$ and $\hat{X}$ to algebraic variables. Then, if we have the Hamiltonian $\hat{\mathcal{H}} = \left(\hat{P} + \hat{X}\right)^2$, we talk of multiplying this out to give $\hat{\mathcal{H}} = (\hat{P})^2 + \hat{P}\cdot\hat{X} + \hat{X}\cdot\hat{P} + (\hat{X})^2$. However, in these expressions we are dealing with operators which can subsequently be applied to wave functions.

We say that a multiplication operator *acts from right to left* if we add new multipliers onto the left. For instance, one might say "multiply $\hat{\mathcal{H}}$ by $\hat{X}$", which would mean $\hat{X}\cdot\hat{\mathcal{H}}$. Conversely, a multiplication operator *acts from left to right* if we add new multipliers onto the right. For instance, an algebraist might say $f\,g$ followed by h is $f\,g\,h$.

These differences in "right to left" or "left to right" only become apparent when our expressions act on some object. For instance the operator $\hat{X}\cdot\hat{\mathcal{H}}$ might act on $\psi(x)$ to give $\left(\hat{X}\cdot\hat{\mathcal{H}}\right)(\psi(x))$. A physicist would *never* write this as $(\psi(x))\left(\hat{X}\cdot\hat{\mathcal{H}}\right)$. Moreover $\left(\hat{X}\cdot\hat{\mathcal{H}}\right)(\psi(x)) = \hat{X}\left(\hat{\mathcal{H}}(\psi(x))\right)$, so whenever we act on a wavefunction with an operator, we do this on the left. It is only through the interaction of the expressions (or operators) being multiplied and their action on the things they can operate on that we get any sense of *handedness*, that is a "right to left" or "left to right" action.

We must address the distinction between "right to left" and "left to right" because we are working so extensively with permutations, and many algebraists and indeed computer algebra programs adopt the "left to right" convention when working with permutations. Examples of such programs are Maple, Magma and GAP. In contrast, *Mathematica*, in the *Combinatorica* package, adopts the "right to left" convention. To give an illustration of this difference,

consider two non-commuting permutations, say (1, 2, 3) and (1, 2) in cycle form. Let us illustrate the typical algebraist's approach of "left to right", via GAP.

```
gap>(1,2,3)*(1,2);
(2,3)
gap> (1,2)*(1,2,3);
(1,3)
```

The full permutations corresponding to these cycles are given by the following.

```
In[81]:= σ1 = FromCycles[{{1, 2, 3}}]

Out[81]= {2, 3, 1}

In[82]:= σ2 = FromCycles[{{1, 2}, {3}}]

Out[82]= {2, 1, 3}
```

As opposed to GAP, *Mathematica* essentially uses "right to left" multiplication as is demonstrated by the following.

```
In[83]:= Permute[σ1, σ2] // ToCycles

Out[83]= {{3, 1}, {2}}
```

This last answer is just the *Mathematica* form of the cycle (1, 3).

After this preamble we can now discuss how and why we have described the action $*_c$ as we have. Since our permuting and relabeling group action, $*_c$, is fundamentally based on our permuting group action, $*_\pi$, it is sufficient to discuss the latter action. We would like our group action to be of the form $* : \mathcal{G} \times \Lambda \to \Lambda$ where Λ is some set of objects. This means that our group operation for $\mathcal{G}$ should be "right to left". Of course, the handedness of our group action and hence of our group operation will only become important once we start using the group to act on objects. We have chosen the "right to left" handedness since it is more in keeping with programming language notations, analysis, and physics, all of which have been intimately involved in the development of the "overall algorithm". Let us now examine several of the consequences of having a right to left operation.

The usual representation of elements of S_n is via bijective functions on the set of integers $\Omega = \{1, \ldots, n\}$. The result we are about to state is such an obvious one that it is often omitted in discussions about permutation groups. However, since the purpose of this appendix is to be strictly clear about the underlying fundamentals of our permutation groups, let us nevertheless state this result. With respect to the group operation of composition, $\circ$, there is a natural group action of permutations on points, and this action is just application. If we call this natural action $*_n$, then $*_n : S_n \times \Omega \to \Omega$, where if $\rho \in S_n$ and $\beta \in \Omega$ then $\rho *_n \beta = \rho(\beta)$. This action is a group action since if $\rho, \sigma \in S_n$ and $\beta \in \Omega$ then

$$\rho *_n (\sigma *_n \beta) = \rho *_n \sigma(\beta) = \rho(\sigma(\beta)) = (\rho \circ \sigma)(\beta) = (\rho \circ \sigma) *_n \beta \qquad \text{(C.1.a)}$$

This natural action is just the way all authors have almost always used permutations and permutation groups. In a somewhat similar manner, in §6.3.1 *Permutations and Configurations*

we have defined $*_\pi : \mathcal{G} \times C \to C$ in *Mathematica*. The question is, in defining the operator as we have, what choices were made and why?

The function `Permute` is implemented in a low-level and extremely efficient way in *Mathematica*, using the function that extracts parts of an expression, specifically `Permute`$[\ell, \sigma] = \ell[\![\sigma]\!]$. So, as was mentioned in §6.3.3, when permuting a list ℓ by a permutation σ, *Mathematica* yields the list $\ell' =$ `Permute`$[\ell, \sigma]$, according to $\ell'_i = \ell_{\sigma(i)}$. Unfortunately, ignoring signs, if we were to define $\sigma *_\pi \ell =$ `Permute`$[\ell, \sigma]$, then $*_\pi$ would not be a group action with respect to $\circ$. In fact, we would get $\rho *_\pi (\sigma *_\pi \ell) = (\sigma \circ \rho) *_\pi \ell$, since their i^{th} components would be equal, as shown by

$$(\rho *_\pi (\sigma *_\pi \ell))_i = (\sigma *_\pi \ell)_{\rho(i)} = \ell_{\sigma(\rho(i))} = \ell_{(\sigma\circ\rho)(i)} = ((\sigma \circ \rho) *_\pi \ell)_i$$

The problem is due to the fact that when permuting a list ℓ by a permutation σ, *Mathematica* uses `Permute`$[\ell, \sigma]$, which looks like a "left to right" type notation, whereas when multiplying permutation ρ by σ, *Mathematica* uses `Permute`$[\rho, \sigma]$, which equals $\rho\circ\sigma$, hence is using a "right to left" notation.

There are two simple ways one can resolve the above difficulty. One is to drop our requirement that our group operation be $\circ$ and define a new binary operator, $\cdot$, by $\sigma \cdot \rho = \rho \circ \sigma$ — resulting in what MacLane[216] calls the *opposite operation* to the group operation $\circ$. In this case, $*_\pi$ becomes a group action with respect to $\cdot$ and, as desired, $\cdot$ is a "left to right" operation. The other solution is to define $*_\pi$ as we did in §6.3.1 *Permutations and Configurations*, namely by $\sigma *_\pi \ell =$ `Permute`$[\ell, \sigma^{-1}]$, in which case $*_\pi$ is a group action with respect to $\circ$. A proof of this fact was given in §6.3.2 *Permutation Groups and Actions*. Another proof that $\rho *_\pi (\sigma *_\pi \ell) = (\rho \circ \sigma) *_\pi \ell$, avoiding the explicit mention of `Permute`, is as follows. When $\ell' =$ `Permute`$[\ell, \sigma]$, we had $\ell'_i = \ell_{\sigma(i)}$. Consequently, when we permute via the inverse permutation, that is $\ell' =$ `Permute`$[\ell, \sigma^{-1}]$, we obtain $\ell'_i = \ell_{\sigma^{-1}(i)}$. So

$$(\rho *_\pi (\sigma *_\pi \ell))_i = (\sigma *_\pi \ell)_{\rho^{-1}(i)} = \ell_{\sigma^{-1}(\rho^{-1}(i))} = \ell_{(\sigma^{-1}\circ\rho^{-1})(i)} = \ell_{(\rho\circ\sigma)^{-1}(i)} = ((\rho \circ \sigma) *_\pi \ell)_i.$$

In summary, we chose a "right to left" operation because it was more in keeping with the setting in which it would be used. And we chose to `Permute` via the inverse of the permutation since then $*_\pi$ is a group action with respect to function composition (actually, a slight extension of function composition to signed function composition). Though in any case, the other conventions we could have chosen are, in a sense, equally valid since they would lead to isomorphic results.

C.2 Evidence for Steepest Descent Conjecture

This appendix provides *extremely strong* evidence that Conjecture 6.9.A, and its extensions to Conjecture 6.12.A and Conjecture 6.13.B, are true. For reference purposes, we repeat below the three conjectures.

Conjecture 6.9.A: Given a signed configuration $a \in C$ and a set of adjacent transpositions $\mathcal{T}$ such that $\forall\, \tau \in \mathcal{T},\ a \preccurlyeq_c \tau *_c a$, then $\theta_{\mathcal{T}}(a) = a$.

Conjecture 6.12.A: Given a signed configuration $a \in C$, which can include indices with fixed elevations, and a set of adjacent transpositions $\mathcal{T}$ such that $\forall\, \tau \in \mathcal{T}\ a \preccurlyeq_c \tau *_c a$, then $\theta_{\mathcal{T}}(a) = a$.

Conjecture 6.13.B: Given a signed configuration $a \in C$, which can include indices with fixed elevations and/or index class extensions, and a set of adjacent transpositions $\mathcal{T}$ such that:

(*i*) $\forall\, \tau \in \mathcal{T}\ a \preccurlyeq_c \tau *_c a$, and

(*ii*) $\forall\, \tau, \overline{\tau} \in \mathcal{T}\ a \preccurlyeq_c \overline{\tau} *_c \tau *_c a$ (where τ, $\overline{\tau}$ are complementary-effect transposition pairs),

then $\theta_{\mathcal{T}}(a) = a$.

These conjectures, in essence, all mean that one can follow a steepest descent method to find the transpositionally canonical or transpositionally minimum configuration after any type of permutation on a given configuration. That is, formally, a locally minimum configuration obtained via any descent path based on the use of the reducing swap operator, defined in §6.9.2 *ReducingSwapQ*, is actually a globally minimum configuration (over the $\mathcal{T}$-equivalence class of configurations). These concepts were explained in §6.9 *Transpositional Canonicalization*.

This section proceeds by first developing some code to investigate the conjecture. Then, utilizing this code, we just exhaustively verify the conjecture for all configurations up to n indices. Due to combinatorial explosion, the practical verification of our conjectures is unfortunately limited to around at most 12 indices.

C.2.1 Transpositional Canonicalization Validation

We need a testing function to validate that our transpositional canonicalization algorithm, presented in §6.9 *Transpositional Canonicalization*, indeed yields the same result as does basic canonicalization with respect to the transpositional generators.

```
In[1]:= transpositionallyCanonicalizeValidQ[
          signedConfiguration_signedConfiguration,
          generators_List] := Θ_generators[signedConfiguration] ==_c
          Min_c[generateConfigurations[signedConfiguration, generators]]
```

We will see that there exists specific sets of transpositions and specific configurations for which our transpositional canonicalization will not always yield the transpositionally minimum configuration. Yet, such instances lie outside all the types of problems that arise in the fields of usage known to the author. Furthermore, any tensor can be rearranged into a tensor which obeys the restrictions needed to ensure that our transpositional canonicalization method is valid — these concepts were explained in §6.8.5 *Jointly Recursively Directional and Extrema Stabilizing (JRDES) Sets*. Therefore, in practice there is no limitation.

We will also be testing our conjectures for the cases when our configurations have indices with fixed elevations. Therefore, we must be able to check if two configurations are essentially the same. This occurs if they are equal up to occurrences of fixed elevations; and that when the configurations differ, say as when we have $s^{\uparrow}_{i,j}$ or $s^{\downarrow}_{i,j}$ in one configuration and $s_{i,j}$ in the other, then the fixed elevation in the former agrees with the natural elevation in the latter. We say that they are essentially the same since when the configurations are reconstituted they will both lead to exactly the same tensor product. Formally, two configurations are essentially equal if and only if they are equal in as far as our ordering will distinguish them. That is why we have defined equality via $==_c$ rather than $\equiv$ in `transpositionallyCanonicalizeValidQ`.

Here is a simple example of these ideas in practice.

```
In[2]:= S_T = {(1 ↔ 2)_+, (3 ↔ 4)_+}
Out[2]= {(1 ↔ 2)_+, (3 ↔ 4)_+}
```

```
In[3]:= Θ_{S_T}[{s^↑_{1,4}, s_{2,3}, s_{3,2}, s^↓_{4,1}}_+]
```

$$\text{Out[3]= } \overline{\{s_{1,3},\ s^{\uparrow}_{2,4},\ s_{3,1},\ s^{\downarrow}_{4,2}\}}_{+}$$

The following two distinct configurations are minimal, and yet essentially the same.

```
In[4]:= Θ_{S_T}[{s_{1,3}, s^↑_{2,4}, s_{3,1}, s^↓_{4,2}}_+]
```

$$\text{Out[4]= } \overline{\{s_{1,3},\ s^{\uparrow}_{2,4},\ s_{3,1},\ s^{\downarrow}_{4,2}\}}_{+}$$

```
In[5]:= Θ_{S_T}[{s^↑_{1,3}, s_{2,4}, s^↓_{3,1}, s_{4,2}}_+]
```

```
Out[5]= {s^↑_{1,3}, s_{2,4}, s^↓_{3,1}, s_{4,2}}_+

In[6]:= {s^↑_{1,3}, s_{2,4}, s^↓_{3,1}, s_{4,2}}_+ ==_c {s_{1,3}, s^↑_{2,4}, s_{3,1}, s^↓_{4,2}}_+

Out[6]= True

In[7]:= transpositionallyCanonicalizeValidQ[{s^↑_{1,4}, s_{2,3}, s_{3,2}, s^↓_{4,1}}_+, S_T]

Out[7]= True
```

C.2.2 Initial Labeling

Let us introduce a specialized function for initially labeling an unlabeled signed configuration. This is a simplified version of our routine `encodeTensors`.

```
In[8]:= initialLabeling @ config_sign_ :=
          MapIndexed[renameIndex[index, position, Position[config, index]] &_{index,position}, conf

In[9]:= renameIndex[index_, position_, allPositions_] := index /; allPositions_len == 1
        renameIndex[index_, position_, allPositions_] :=
          s[position_[[1]], (Flatten[allPositions] \ position)_[[1]]] /; allPositions_len == 2
```

Here is an example using `initialLabeling`.

```
In[11]:= initialLabeling[{a, b, b, a}_+]

Out[11]= {s_{1,4}, s_{2,3}, s_{3,2}, s_{4,1}}_+
```

C.2.3 Generating all Configurations

In this appendix, to demonstrate the properties of the transpositional canonicalization operator, we will need to perform tests on all possible configurations of a given number of indices which are all summed. There are several ways one could generate all such configurations. The first and most obvious way is to create all permutations of some initial set, relabel these, and discard any duplicates. This is easy to implement. For example, the following creates every configuration on four indices with each index being summed.

```
In[12]:= config_+

Out[12]= config_+

In[13]:= Union[initialLabeling /@ (config_+ &_config /@ Permutations[{a, a, b, b}])]

Out[13]= {{s_{1,2}, s_{2,1}, s_{3,4}, s_{4,3}}_+, {s_{1,3}, s_{2,4}, s_{3,1}, s_{4,2}}_+, {s_{1,4}, s_{2,3}, s_{3,2}, s_{4,1}}_+}
```

It is easy to generalize the method above; however, the above algorithm's space and time efficiency is awful. For example, to generate all configurations on 10 indices requires calculating roughly 3.6 million configurations even though there are only 945 different configurations.

Therefore, due to practical considerations, we must turn to a different algorithm to calculate the configurations.

The second way we can generate all configurations is by using a recursive method. Basically, for each configuration on n indices, we create $n+1$ configurations by inserting the symbol a into the original configuration at every possible position. For each of these configurations, we then create $n+2$ configurations by repeating the above insertion process. We then relabel all these configurations. The resulting set will span the complete set of configurations on $n+2$ indices. In this way we can create all configurations on $n+2$ indices from all the configurations on n indices. Let us formally present the recursive algorithm that does this.

```
In[14]:= allConfigurations @ 2 := {{s1,2, s2,1}+};
         allConfigurations @ n_Integer?EvenQ :=
           Union[initialLabeling /@
                 (insertSymbolA @ insertSymbolA @ allConfigurations[n - 2] //.
                       si_,j_ :> Min[i, j])];
```

This code uses `insertSymbolA`, which simply creates configurations with a single a added to them in every possible position.

```
In[16]:= insertSymbolA @ signedConfigurations_List :=
           Flatten[
             Table[Insert[sc, a, {2, i}], {i, 1, 1 + sc[[2]]len}] &sc /@ signedConfigurations]
```

There are some configurations that are generated twice this way; however, all duplicate configurations are removed in the end by taking the union of the configurations. We can easily test this method to see that it is giving the correct results by comparing it to the complete set of configurations on four indices generated above.

```
In[17]:= allConfigurations @ 4
Out[17]= {{s1,2, s2,1, s3,4, s4,3}+, {s1,3, s2,4, s3,1, s4,2}+, {s1,4, s2,3, s3,2, s4,1}+}
```

To get some idea of the growth in size and complexity, examine the following.

```
In[18]:= Table[allConfigurations[i]len, {i, 4, 10, 2}]
Out[18]= {3, 15, 105, 945}
```

With a small amount of insight, this can be recognized as the following sequence.

```
In[19]:= Table[i! / ((i / 2)! 2^(i/2)), {i, 4, 10, 2}]
Out[19]= {3, 15, 105, 945}
```

C.2.4 Generating all Configurations with Fixed Elevations

Due to considerations of length, the following code will not be explained in detail. Interested readers can trace through the code for themselves.

```
In[20]:= allConfigurationsWithElevations @ n_Integer?EvenQ :=
           Flatten[allElevations /@ allConfigurations @ n]

In[21]:= allElevations [sc : config_sign_] :=
           Block[{elevationSubsets, configChanges},
             elevationSubsets =
               Drop[Sort @ Subsets @ Cases[config, s_i_,j_ /; i < j], 1] /.
                 s_i_,j_ → {{s↑i,j}, {s↓i,j}};
             configChanges = Flatten[(Outer[Join, Sequence @@ #, 1] & /@
                               elevationSubsets) /. {a__s} → changeElevations @ a];
             ((sc /. elevationToRules @ # &) /@ configChanges +{} {sc})]

In[22]:= elevationToRules @ aChange_changeElevations :=
           aChange /. {s↑i_,j_ :↦ {si,j → s↑i,j, sj,i → s↓j,i},
                       s↓i_,j_ :↦ {si,j → s↓i,j, sj,i → s↑j,i}} /.
             changeElevations @ any__ :↦ Flatten @ {any}
```

Let us quickly demonstrate how this code works in practice.

```
In[23]:= allConfigurations @ 4

Out[23]= {{s1,2, s2,1, s3,4, s4,3}+, {s1,3, s2,4, s3,1, s4,2}+, {s1,4, s2,3, s3,2, s4,1}+}
```

We generate all possible elevations on a signed configuration as follows.

```
In[24]:= allElevations @ {s1,3, s2,4, s3,1, s4,2}+

Out[24]= {{s↑1,3, s2,4, s↓3,1, s4,2}+, {s↓1,3, s2,4, s↑3,1, s4,2}+, {s1,3, s↑2,4, s3,1, s↓4,2}+,
          {s1,3, s↓2,4, s3,1, s↑4,2}+, {s↑1,3, s↑2,4, s↓3,1, s↓4,2}+,
          {s↑1,3, s↓2,4, s↓3,1, s↑4,2}+, {s↓1,3, s↑2,4, s↑3,1, s↓4,2}+,
          {s↓1,3, s↓2,4, s↑3,1, s↑4,2}+, {s1,3, s2,4, s3,1, s4,2}+}
```

With respect to a configuration of n indices, by combining the generation of all configurations with the generation of all elevations for a configuration we can generate all configurations with fixed elevations.

```
In[25]:= allConfigurationsWithElevations @ 4
```

Out[25]= $\{\{s^{\uparrow}_{1,2}, s^{\downarrow}_{2,1}, s_{3,4}, s_{4,3}\}_+, \{s^{\downarrow}_{1,2}, s^{\uparrow}_{2,1}, s_{3,4}, s_{4,3}\}_+, \{s_{1,2}, s_{2,1}, s^{\uparrow}_{3,4}, s^{\downarrow}_{4,3}\}_+,$
$\{s_{1,2}, s_{2,1}, s^{\downarrow}_{3,4}, s^{\uparrow}_{4,3}\}_+, \{s^{\uparrow}_{1,2}, s^{\downarrow}_{2,1}, s^{\uparrow}_{3,4}, s^{\downarrow}_{4,3}\}_+,$
$\{s^{\uparrow}_{1,2}, s^{\downarrow}_{2,1}, s^{\downarrow}_{3,4}, s^{\uparrow}_{4,3}\}_+, \{s^{\downarrow}_{1,2}, s^{\uparrow}_{2,1}, s^{\uparrow}_{3,4}, s^{\downarrow}_{4,3}\}_+,$
$\{s^{\downarrow}_{1,2}, s^{\uparrow}_{2,1}, s^{\downarrow}_{3,4}, s^{\uparrow}_{4,3}\}_+, \{s_{1,2}, s_{2,1}, s_{3,4}, s_{4,3}\}_+,$
$\{s^{\uparrow}_{1,3}, s_{2,4}, s^{\downarrow}_{3,1}, s_{4,2}\}_+, \{s^{\downarrow}_{1,3}, s_{2,4}, s^{\uparrow}_{3,1}, s_{4,2}\}_+,$
$\{s_{1,3}, s^{\uparrow}_{2,4}, s_{3,1}, s^{\downarrow}_{4,2}\}_+, \{s_{1,3}, s^{\downarrow}_{2,4}, s_{3,1}, s^{\uparrow}_{4,2}\}_+,$
$\{s^{\uparrow}_{1,3}, s^{\uparrow}_{2,4}, s^{\downarrow}_{3,1}, s^{\downarrow}_{4,2}\}_+, \{s^{\uparrow}_{1,3}, s^{\downarrow}_{2,4}, s^{\downarrow}_{3,1}, s^{\uparrow}_{4,2}\}_+,$
$\{s^{\downarrow}_{1,3}, s^{\uparrow}_{2,4}, s^{\uparrow}_{3,1}, s^{\downarrow}_{4,2}\}_+, \{s^{\downarrow}_{1,3}, s^{\downarrow}_{2,4}, s^{\uparrow}_{3,1}, s^{\uparrow}_{4,2}\}_+,$
$\{s_{1,3}, s_{2,4}, s_{3,1}, s_{4,2}\}_+, \{s^{\uparrow}_{1,4}, s_{2,3}, s_{3,2}, s^{\downarrow}_{4,1}\}_+,$
$\{s^{\downarrow}_{1,4}, s_{2,3}, s_{3,2}, s^{\uparrow}_{4,1}\}_+, \{s_{1,4}, s^{\uparrow}_{2,3}, s^{\downarrow}_{3,2}, s_{4,1}\}_+,$
$\{s_{1,4}, s^{\downarrow}_{2,3}, s^{\uparrow}_{3,2}, s_{4,1}\}_+, \{s^{\uparrow}_{1,4}, s^{\uparrow}_{2,3}, s^{\downarrow}_{3,2}, s^{\downarrow}_{4,1}\}_+,$
$\{s^{\uparrow}_{1,4}, s^{\downarrow}_{2,3}, s^{\uparrow}_{3,2}, s^{\downarrow}_{4,1}\}_+, \{s^{\downarrow}_{1,4}, s^{\uparrow}_{2,3}, s^{\downarrow}_{3,2}, s^{\uparrow}_{4,1}\}_+,$
$\{s^{\downarrow}_{1,4}, s^{\downarrow}_{2,3}, s^{\uparrow}_{3,2}, s^{\uparrow}_{4,1}\}_+, \{s_{1,4}, s_{2,3}, s_{3,2}, s_{4,1}\}_+\}$

C.2.5 Generation all Configurations with Class Extended Indices

We need to be able to generate all possible configurations involving all possible class extended indices. This can be accomplished by adding class extensions to the dummy indices of pre-existing configurations.

Initially we use numbers to represent the different index classes. Thus, at first, we will represent a given set of class extensions by a list of numbers. For instance, {1,2,2,1} would mean the first pair of summed indices of a configuration is of class 1, the second pair is of class 2, the third pair is of class 2, and the fourth pair is of class 1. Here is a function which will generate all possible class extension representations.

```
In[26]:= allClassExtensionRepresentations @ n_Integer := Select[
           Subsets @ Flatten @ Join @ Table[Range @ n, {n}], subset_len == n &_subset]
```

Some of the representations generated by `allClassExtensionRepresentations` are essentially equivalent. For instance, {3,4,4,3} is essentially the same as {1,2,2,1}. The following function, `normalizeRepresentation`, just normalizes all class representations.

```
In[27]:= normalizeRepresentation @ list_List := Fold[lower, list, Range @ list_len]
         lower[list : {___, a_, ___}, a_] := list
         lower[list_List, n_] /; Select[list, i ≥ n &_i] ≠ {} :=
            lower[If[i > n, i - 1, i] &_i /@ list, n]
         lower[list_List, n_Integer] := list
```

Here are all possible variations on the class extensions for 8 indices. (Both indices of a summed pair must obviously be of the same class.)

```
In[31]:= Union[normalizeRepresentation /@ allClassExtensionRepresentations[4]]
```

```
Out[31]= {{1, 1, 1, 1}, {1, 1, 1, 2}, {1, 1, 2, 1}, {1, 1, 2, 2}, {1, 1, 2, 3},
          {1, 1, 3, 2}, {1, 2, 1, 1}, {1, 2, 1, 2}, {1, 2, 1, 3}, {1, 2, 2, 1},
          {1, 2, 2, 2}, {1, 2, 2, 3}, {1, 2, 3, 1}, {1, 2, 3, 2}, {1, 2, 3, 3},
          {1, 2, 3, 4}, {1, 2, 4, 3}, {1, 3, 1, 2}, {1, 3, 2, 1}, {1, 3, 2, 2},
          {1, 3, 2, 3}, {1, 3, 2, 4}, {1, 3, 3, 2}, {1, 3, 4, 2}, {1, 4, 2, 3},
          {1, 4, 3, 2}, {2, 1, 1, 1}, {2, 1, 1, 2}, {2, 1, 1, 3}, {2, 1, 2, 1},
          {2, 1, 2, 2}, {2, 1, 2, 3}, {2, 1, 3, 1}, {2, 1, 3, 2}, {2, 1, 3, 3},
          {2, 1, 3, 4}, {2, 1, 4, 3}, {2, 2, 1, 1}, {2, 2, 1, 2}, {2, 2, 1, 3},
          {2, 2, 2, 1}, {2, 2, 3, 1}, {2, 3, 1, 1}, {2, 3, 1, 2}, {2, 3, 1, 3},
          {2, 3, 1, 4}, {2, 3, 2, 1}, {2, 3, 3, 1}, {2, 3, 4, 1}, {2, 4, 1, 3},
          {2, 4, 3, 1}, {3, 1, 1, 2}, {3, 1, 2, 1}, {3, 1, 2, 2}, {3, 1, 2, 3},
          {3, 1, 2, 4}, {3, 1, 3, 2}, {3, 1, 4, 2}, {3, 2, 1, 1}, {3, 2, 1, 2},
          {3, 2, 1, 3}, {3, 2, 1, 4}, {3, 2, 2, 1}, {3, 2, 3, 1}, {3, 2, 4, 1},
          {3, 3, 1, 2}, {3, 3, 2, 1}, {3, 4, 1, 2}, {3, 4, 2, 1}, {4, 1, 2, 3},
          {4, 1, 3, 2}, {4, 2, 1, 3}, {4, 2, 3, 1}, {4, 3, 1, 2}, {4, 3, 2, 1}}
```

Having generated all numerical representations of a class, we subsequently transform these into real classes.

```
In[32]:= toClasses @ classes_ :=
           (Union @ classes) /. i_Integer :> Symbol [ "class" <> ToString @ i]
```

Here is a simple way to say that the following are valid index classes. (We never consider testing 14 index configurations in any event due to combinatorial explosion.)

```
In[33]:= ValidIndexClassQ[class1] = True;
         ValidIndexClassQ[class2] = True;
         ValidIndexClassQ[class3] = True;
         ValidIndexClassQ[class4] = True;
         ValidIndexClassQ[class5] = True;
         ValidIndexClassQ[class6] = True;
```

We need to be able to affix the class extensions to the indices at the levels required.

```
In[39]:= mergeClasses[sc_, {}] := sc
         mergeClasses[
             sc : {___, index : (s↑i_,j_ | s↓i_,j_ | si_,j_), ___}sign_, {class_, rest___}] :=
           mergeClasses[MapAt[setClass[sij, class] &sij, sc, {{2, i}, {2, j}}], {rest}]
```

$$\texttt{In[41]:=}\ \texttt{setClass}[s^{\uparrow}_{i\_,j\_},\ class\_] := s^{class,\uparrow}_{i,j}$$

$$\texttt{setClass}[s^{\downarrow}_{i\_,j\_},\ class\_] := s^{class,\downarrow}_{i,j}$$

$$\texttt{setClass}[s_{i\_,j\_},\ class\_] := s^{class}_{i,j}$$

Here is an example of `mergeClasses`.

$$\texttt{In[44]:=}\ \texttt{mergeClasses}\left[\overline{\{s^{\uparrow}_{1,4},\ s_{2,3},\ s_{3,2},\ s^{\downarrow}_{4,1}\}}_{+},\right.$$
$$\left.\{\texttt{SpaceTimeIndices},\ \texttt{GeneralIndices}\}\right]$$

$$\texttt{Out[44]=}\ \overline{\{st^{\uparrow}_{1,4},\ sg_{2,3},\ sg_{3,2},\ st^{\downarrow}_{4,1}\}}_{+}$$

Finally, we create a simple function that generates all possible class extended versions of a given signed configuration.

```
In[45]:= addAllClasses @ sc_signedConfiguration :=
           mergeClasses[sc, extensions] &extensions /@
             toClasses[normalizeRepresentation /@
                 allClassExtensionRepresentations @
                   (Count[sc, s @ __, {2}] / 2)]

In[46]:= addAllClasses @ scs_List := Flatten [ addAllClasses /@ scs]

In[47]:= addAllClasses @ {s1,4↑, s2,3, s3,2, s4,1↓}+
```

$$\text{Out[47]= } \left\{\overline{\{s_{1,4}^{\text{class1},\uparrow}, s_{2,3}^{\text{class1}}, s_{3,2}^{\text{class1}}, s_{4,1}^{\text{class1},\downarrow}\}_+}, \overline{\{s_{1,4}^{\text{class1},\uparrow}, s_{2,3}^{\text{class2}}, s_{3,2}^{\text{class2}}, s_{4,1}^{\text{class1},\downarrow}\}_+}, \overline{\{s_{1,4}^{\text{class2},\uparrow}, s_{2,3}^{\text{class1}}, s_{3,2}^{\text{class1}}, s_{4,1}^{\text{class2},\downarrow}\}_+}\right\}$$

These configurations can become somewhat involved, as evidenced by the following.

```
In[48]:= signedConfigurations = addAllClasses @ allConfigurationsWithElevations @ 6;
         RandomKSubset[signedConfigurations, 5]
```

$$\text{Out[49]= } \left\{\begin{array}{l}\overline{\{s_{1,2}^{\text{class2},\uparrow}, s_{2,1}^{\text{class2},\downarrow}, s_{3,4}^{\text{class1}}, s_{4,3}^{\text{class1}}, s_{5,6}^{\text{class3}}, s_{6,5}^{\text{class3}}\}_+},\\ \overline{\{s_{1,3}^{\text{class1}}, s_{2,4}^{\text{class1}}, s_{3,1}^{\text{class1}}, s_{4,2}^{\text{class1}}, s_{5,6}^{\text{class1},\downarrow}, s_{6,5}^{\text{class1},\uparrow}\}_+},\\ \overline{\{s_{1,3}^{\text{class2},\downarrow}, s_{2,5}^{\text{class1},\uparrow}, s_{3,1}^{\text{class2},\uparrow}, s_{4,6}^{\text{class1},\uparrow}, s_{5,2}^{\text{class1},\downarrow}, s_{6,4}^{\text{class1},\downarrow}\}_+},\\ \overline{\{s_{1,6}^{\text{class3},\uparrow}, s_{2,3}^{\text{class2}}, s_{3,2}^{\text{class2}}, s_{4,5}^{\text{class1}}, s_{5,4}^{\text{class1}}, s_{6,1}^{\text{class3},\downarrow}\}_+},\\ \overline{\{s_{1,6}^{\text{class2},\uparrow}, s_{2,5}^{\text{class3},\downarrow}, s_{3,4}^{\text{class1},\uparrow}, s_{4,3}^{\text{class1},\downarrow}, s_{5,2}^{\text{class3},\uparrow}, s_{6,1}^{\text{class2},\downarrow}\}_+}\end{array}\right\}$$

The number of possible configurations when including class extensions and fixed elevations exhibits combinatorial explosion. For example, there are just 15 configurations on 6 indices if class extensions and fixed elevations are not included, but 5265 if they are.

```
In[50]:= signedConfigurations ten

Out[50]= 5265
```

C.2.6 Restriction: No Disconnected Transpositional Orbits

To analyze the following algorithms, it is important to be able to get some idea of how a set of permutations can act on a configuration. If we borrow the terminology from group action, we can examine the "orbits" of a permutation. For instance, the transpositions $(1 \leftrightarrow 4)_+$, $(2 \leftrightarrow 5)_+$, and $(5 \leftrightarrow 6)_+$ can move an index between 1 and 4 or move an index between 2, 5 and 6. We can represent these orbits by lists as follows: {■,□,□,■,□,□} and {□,■,□,□,■,■}. Alternatively, we can include all the orbits into the same list by labeling each orbit by its smallest element: {1,2,□,1,2,2}.

Definition 13.2.A: An *orbit* of a set of generators $\mathcal{S}$ is a set of positions where an index β in any one of the positions can be moved into any other position in the set by applying a sequence of the generators. Basically an orbit of a set of generators is just the set $\beta^{\langle \mathcal{S} \rangle}$.

An orbit is *connected* if there are no gaps in the orbit. For instance, an orbit of the shape {□,■,■,■,□,□} would be connected, while an orbit of the shape {□,■,□,□,■,■} would be *disconnected*. Note, we should be cognizant of the distinction between the orbit of an index under a set of permutations, and the orbit of the configuration under the permutation group action, that is the orbit of all equivalent configurations under the permutation symmetries.

We can now ask the question that was raised when we first looked at transpositional canonicalization. When will our steepest descent method of transpositional canonicalization actually yield a minimum transpositional equivalence class representative? The following example demonstrates that one must not have disconnected transpositions that cross each other, like $(1 \leftrightarrow 4)_+$ and $(2 \leftrightarrow 5)_+$. Our transpositional canonicalization algorithm will frequently fail on different configurations with crossed disconnected transpositional generators.

```
In[51]:= S_τ = {(1 ↔ 4)_+, (2 ↔ 5)_+};
         testConfiguration = {s_{1,5}, s_{2,3}, s_{3,2}, s_{4,6}, s_{5,1}, s_{6,4}}_+;

In[53]:= Θ_{S_τ}[testConfiguration]

Out[53]= {s_{1,5}, s_{2,3}, s_{3,2}, s_{4,6}, s_{5,1}, s_{6,4}}_+

In[54]:= generateConfigurations[testConfiguration, S_τ]

Out[54]= {{s_{1,2}, s_{2,1}, s_{3,5}, s_{4,6}, s_{5,3}, s_{6,4}}_+, {s_{1,5}, s_{2,3}, s_{3,2}, s_{4,6}, s_{5,1}, s_{6,4}}_+,
           {s_{1,6}, s_{2,3}, s_{3,2}, s_{4,5}, s_{5,4}, s_{6,1}}_+, {s_{1,6}, s_{2,4}, s_{3,5}, s_{4,2}, s_{5,3}, s_{6,1}}_+}

In[55]:= Min_c[%]

Out[55]= {s_{1,6}, s_{2,4}, s_{3,5}, s_{4,2}, s_{5,3}, s_{6,1}}_+
```

However, even if the transpositional orbits do not cross each other, being disconnected is sometimes enough to make transpositional canonicalization fail. Typically this requires at least two disconnections in orbits and does not occur in situations other then when an orbit has a disconnected singleton part. (In other words, this occurs very rarely, but it can happen.) In fact, for six indices there are just 12 cases out of 1323 different combinations of transpositions and configurations for which this will occur. Here is one of the examples where transpositional canonicalization fails.

```
In[56]:= S_τ = {(1 ↔ 3)_+, (4 ↔ 6)_+};
         testConfiguration = {s_{1,5}, s_{2,4}, s_{3,6}, s_{4,2}, s_{5,1}, s_{6,3}}_+;

In[58]:= Θ_{S_τ}[testConfiguration]

Out[58]= {s_{1,5}, s_{2,4}, s_{3,6}, s_{4,2}, s_{5,1}, s_{6,3}}_+

In[59]:= generateConfigurations[testConfiguration, S_τ]

Out[59]= {{s_{1,4}, s_{2,6}, s_{3,5}, s_{4,1}, s_{5,3}, s_{6,2}}_+, {s_{1,5}, s_{2,4}, s_{3,6}, s_{4,2}, s_{5,1}, s_{6,3}}_+,
           {s_{1,5}, s_{2,6}, s_{3,4}, s_{4,3}, s_{5,1}, s_{6,2}}_+, {s_{1,6}, s_{2,4}, s_{3,5}, s_{4,2}, s_{5,3}, s_{6,1}}_+}

In[60]:= Min_c[%]

Out[60]= {s_{1,4}, s_{2,6}, s_{3,5}, s_{4,1}, s_{5,3}, s_{6,2}}_+
```

C.2.7 Design Consequences

At this stage we can now see the underlying choices faced when the algorithm was being created. For this particular facet of the overall problem, we had a dilemma with two alternatives: (*i*) we could allow disconnected orbits that do not overlap, and try to find which class of transpositional generators will work with our transpositional canonicalization algorithm, or (*ii*), we could only allow connected orbits. In practice the author has only seen tensors where the transpositional orbits are connected. For instance, this is true of Christoffel symbols, Riemann tensors, Stress energy tensors, vector potentials, Maxwell tensors, Levi-Civita tensor densities, etc. However, it is a virtual certainty that in some branch of physics, someone somewhere at sometime has defined a tensor which has disconnected transpositional orbits. In this case to use transpositional canonicalization, they must introduce an equivalent intermediate tensor which has connected orbits.

C.2.8 The Construction of all Valid Transpositional Generating Sets

We now have a possible criterion on the set of transpositions under which our steepest descent algorithm will work — that is, we hypothesis that our generating sets must be adjacent in order to ensure that transpositional canonicalization works. We would like to demonstrate the hypothesis by generating all possible sets of transpositions fulfilling the criterion and showing that every one of these sets of transpositions obeys the transpositional canonicalization property on every possible configuration on a fixed number of indices.

Since we will only use adjacent transpositions, and hence always have connected orbits, we can obtain all valid transpositional generator sets on n indices simply by the following.

```
In[61]:= allValidTranspositionSets @ n_Integer?Positive :=
           Subsets[Table[(i ↔ i + 1)_+, {i, 1, n - 1}]] \ {{}}
```

Technical Note: Prior to the final choice published herein, I experimented with several different types of orbit generation schemes, for several allowed classes of orbits. The code is available upon request.

We can combine the complexity orders of both the generation of configurations and the different generating sets to arrive at the number of tests that must be done on each number of indices.

```
In[62]:= Table[ i ! / ((i / 2) ! 2^(i/2)) (2^(i-1) - 1), {i, 4, 14, 2}]

Out[62]= {21, 465, 13335, 482895, 21278565, 1106890785}
```

C.2.9 Testing the Transpositional Canonicalization of Configurations

The following routine is a "smart way" to test that transpositional canonicalization is working. Given a whole class of configurations, we choose one, and generate all transpositionally equivalent configurations. Our transpositional canonicalization operator is then applied to all of these equivalent configurations. If the result of all of these applications is the same configuration (up to equivalence), then we know that on these configurations transpositional canonicalization is valid. We then eliminate this entire equivalence class of configurations from the set of total configurations remaining to be tested, and repeat the whole process.

```
In[63]:= testTranspositionalCanonicalizationOfConfigurations [
             signedConfiguration_signedConfiguration, generators_List] :=
           testTranspositionalCanonicalizationOfConfigurations [
             {signedConfiguration}, generators]

In[64]:= testTranspositionalCanonicalizationOfConfigurations [
             configurationsToTest_List, generators_List] :=
           Block[{
               passed = True,
               configurationsLeft = configurationsToTest,
               transpositionallyEquivalentConfigurations = {}},
             While [configurationsLeft ≠ {} ∧ passed == True,
               transpositionallyEquivalentConfigurations =
                 generateConfigurations [ configurationsLeft[[1]], generators];
               If[ Length @ Union[polishElevations @ Θgenerators [c] &c /@
                                  transpositionallyEquivalentConfigurations] >
                     1, passed = False];
               configurationsLeft = configurationsLeft \ transpositionallyEquivalentConfigurations; ];
             passed]

In[65]:= polishElevations @ configuration_ :=
           configuration /. {s_{i_,j_}^{class_,↑} /; i < j :↦ s_{i,j}^{class}, s_{i_,j_}^{class_,↓} /; i > j :↦ s_{i,j}^{class},
                 s_{i_,j_}^{↑} /; i < j :↦ s_{i,j}, s_{i_,j_}^{↓} /; i > j :↦ s_{i,j}}
```

Thus, given a set of configurations and a set of generating transpositions, we can exhaustively test all possible cases.

```
In[66]:= exhaustiveTranspositionalCanonicalizationTest [
             configurationsToTest_List, generatingTranspositions_List] :=
           Block[{failed = 0, passed = 0, failedList = {}},
             Do[
               If[ testTranspositionalCanonicalizationOfConfigurations [
                     configurationsToTest, generatingTranspositions[[i]] ],
                   ++passed,
                   ++failed; AppendTo[failedList, generatingTranspositions[[i]]]],
               {i, 1, generatingTranspositions_fen}];
             Print["Passed :", passed];
```

```
Print["Failed :", failed];
Print["Failed generating Transposition sets :",
   Union @ failedList];]
```

C.2.10 Example: Testing the Conjecture for 6 Indices

We are now, finally, in a position to verify Conjecture 6.9.A by brute force for up to n indices. In this subsection we will tackle the case $n = 6$. (This also subsumes all cases of n less than 6.) First we generate all configurations on 6 indices, and then generate all possible adjacently generated transpositional generating sets.

```
In[67]:= signedConfigurations = allConfigurations[6];
         generatingTranspositions = allValidTranspositionSets[6];
```

Here are the kind of configurations we are verifying transpositional canonicalization for.

```
In[69]:= RandomKSubset[signedConfigurations, 5]
```

$$\text{Out[69]=}\ \{\overline{\{s_{1,2}, s_{2,1}, s_{3,6}, s_{4,5}, s_{5,4}, s_{6,3}\}}_{+},\ \overline{\{s_{1,3}, s_{2,4}, s_{3,1}, s_{4,2}, s_{5,6}, s_{6,5}\}}_{+},\ \overline{\{s_{1,3}, s_{2,6}, s_{3,1}, s_{4,5}, s_{5,4}, s_{6,2}\}}_{+},\ \overline{\{s_{1,5}, s_{2,3}, s_{3,2}, s_{4,6}, s_{5,1}, s_{6,4}\}}_{+},\ \overline{\{s_{1,6}, s_{2,4}, s_{3,5}, s_{4,2}, s_{5,3}, s_{6,1}\}}_{+}\}$$

There are 15 configurations and 31 different transpositional generating sets for which we must verify the conjecture.

```
In[70]:= signedConfigurations[[ten]]
```

```
Out[70]= 15
```

```
In[71]:= generatingTranspositions[[ten]]
```

```
Out[71]= 31
```

```
In[72]:= exhaustiveTranspositionalCanonicalizationTest[
            signedConfigurations, generatingTranspositions] // Timing

         Passed :31

         Failed :0

         Failed generating Transposition sets :{}

Out[72]= {1.73333 Second, Null}
```

Therefore we have verified, by brute force, that transpositional canonicalization will work correctly with any set of adjacent transpositions and any standard configuration with up to 6 standard indices.

C.2.11 Optimizing the Testing of Transpositional Canonicalization

Actually, we can make a significant optimization to our brute force testing algorithm. Starting with the set of all possible configurations, we can set all indices not involved in our calculations to fixed high indices, say $a1^{\uparrow}$, $a2^{\uparrow}$, $a3^{\uparrow}$,..., and then take the union of this resulting set. This reduces the number of configurations, since we will not be considering all of the multitude of variants of a configuration where only indices that do not affect the calculation are changing. A moment's reflection will confirm that this simplification is valid. Here are the functions we need in order to normalize a list of configurations.

```
In[73]:= normalizeConfigurations[signedConfigurations_List, movedPoints_List] :=
           Block[{uniformConfig = Table[High[ToExpression["a" <> ToString @i], i],
                   {i, 1, signedConfigurations[[1, 2]]ten}]},
             Union[toUniformConfiguration[sc, movedPoints] &sc /@ signedConfigurations]]

In[74]:= toUniformConfiguration[config_sign_, movedPoints_List] :=
           Block[
             {indicesToPreserve = List /@ ((sij[[2]] &sij /@ config[[movedPoints]]) ⋃ movedPoints)},
             ReplacePart[uniformConfig, config, indicesToPreserve, indicesToPreserve]sign]
```

For some of the smaller transpositional generating sets, we can get a rough idea of the reduction in the number of configurations that need to be considered.

```
In[75]:= signedConfigurations = addAllClasses @ allConfigurationsWithElevations[6];

In[76]:= signedConfigurationsten

Out[76]= 5265

In[77]:= reducedSet = normalizeConfigurations[signedConfigurations, {1, 2}];
         reducedSetten

Out[78]= 873
```

Thus we have reduced the number of configurations which we must test by a factor of 6. The normalized configurations look like the following.

```
In[79]:= RandomKSubset[reducedSet, 5]
```

$$\Big\{\{s_{1,3}^{\text{class2}}, s_{2,4}^{\text{class2}}, s_{3,1}^{\text{class2}}, s_{4,2}^{\text{class2}}, a5_5^{\uparrow}, a6_6^{\uparrow}\}_+,$$
$$\{s_{1,3}^{\text{class2}}, s_{2,5}^{\text{class3}}, s_{3,1}^{\text{class2}}, a4_4^{\uparrow}, s_{5,2}^{\text{class3}}, a6_6^{\uparrow}\}_+,$$
$$\{s_{1,3}^{\text{class3}}, s_{2,4}^{\text{class2},\uparrow}, s_{3,1}^{\text{class3}}, s_{4,2}^{\text{class2},\downarrow}, a5_5^{\uparrow}, a6_6^{\uparrow}\}_+,$$
$$\{s_{1,4}^{\text{class2},\uparrow}, s_{2,6}^{\text{class1}}, a3_3^{\uparrow}, s_{4,1}^{\text{class2},\downarrow}, a5_5^{\uparrow}, s_{6,2}^{\text{class1}}\}_+,$$
$$\{s_{1,5}^{\text{class2},\downarrow}, s_{2,4}^{\text{class2},\uparrow}, a3_3^{\uparrow}, s_{4,2}^{\text{class2},\downarrow}, s_{5,1}^{\text{class2},\uparrow}, a6_6^{\uparrow}\}_+\Big\}$$

When most of the indices are "moved", then the reduction in the number of configurations that must be tested is much smaller.

```
In[80]:= reducedSet = normalizeConfigurations[signedConfigurations, {1, 2, 3, 4}];
         reducedSet_ten

Out[81]= 4428
```

Here is the modification to our testing function `testTranspositionalCanonicalization-OfConfigurations`.

```
In[82]:= testTranspositionalCanonicalizationOfConfigurations[
             configurationsToTest_List, generators_List] :=
           Block[{
               passed = True,
               configurationsLeft = normalizeConfigurations[configurationsToTest,
                     Union @ Flatten[st_[[2]] &_st /@ generators]],
               transpositionallyEquivalentConfigurations = {}},
             While[configurationsLeft ≠ {} ∧ passed == True,
               transpositionallyEquivalentConfigurations =
                 generateConfigurations[configurationsLeft_[[1]], generators];
               If[
                 Union[polishElevations @ Θ_generators[c] &_c /@ transpositionallyEquivalentCc
                   > 1, passed = False];
               configurationsLeft = configurationsLeft \ transpositionallyEquivalentConfigurations;];
             passed]

In[83]:= exhaustiveTranspositionalCanonicalizationTest[
             configurationsToTest_List, generatingTranspositions_List] :=
           Block[{failed = 0, passed = 0, failedList = {}},
             Do[
               If[ testTranspositionalCanonicalizationOfConfigurations[
                     configurationsToTest, generatingTranspositions_[[i]]],
                 ++passed,
                 ++failed; AppendTo[failedList, generatingTranspositions_[[i]]]],
               {i, 1, generatingTranspositions_ten}];
             Print["Passed :", passed];
             Print["Failed :", failed];
             Print["Failed generating Transposition sets :",
               Union @ failedList];]
```

Let us now use this optimized testing function for all of our forthcoming brute force verifications.

C.2.12 Example: Testing the Conjecture for 6 Indices with Fixed Elevations

Now that we have presented the general style in which we will proceed, let us verify Conjecture 6.12.A for all configurations having at most 6 indices, including those containing indices of fixed elevation, but excluding any class extensions.

```
In[84]:= signedConfigurations = allConfigurationsWithElevations[6];
         generatingTranspositions = allValidTranspositionSets[6];
```

Here are some examples of the kind of configuration for which we are verifying the transpositional canonicalization conjecture.

```
In[86]:= RandomKSubset[signedConfigurations, 5]
```

Out[86]= $\left\{\overline{\{s^{\uparrow}_{1,4}, s_{2,5}, s_{3,6}, s^{\downarrow}_{4,1}, s_{5,2}, s_{6,3}\}_+}, \overline{\{s_{1,4}, s^{\downarrow}_{2,5}, s^{\downarrow}_{3,6}, s_{4,1}, s^{\uparrow}_{5,2}, s^{\uparrow}_{6,3}\}_+}, \overline{\{s_{1,5}, s^{\uparrow}_{2,6}, s^{\uparrow}_{3,4}, s^{\downarrow}_{4,3}, s_{5,1}, s^{\downarrow}_{6,2}\}_+}, \overline{\{s^{\uparrow}_{1,6}, s_{2,3}, s_{3,2}, s^{\downarrow}_{4,5}, s^{\uparrow}_{5,4}, s^{\downarrow}_{6,1}\}_+}, \overline{\{s_{1,6}, s_{2,4}, s^{\downarrow}_{3,5}, s_{4,2}, s^{\uparrow}_{5,3}, s_{6,1}\}_+}\right\}$

We can see that by including indices with fixed elevations, we have generated dramatically more configurations.

```
In[87]:= signedConfigurations_ten
```

```
Out[87]= 405
```

```
In[88]:= generatingTranspositions_ten
```

```
Out[88]= 31
```

There are 405 configurations and 31 different transpositional generating sets for which we must verify the conjecture.

```
In[89]:= exhaustiveTranspositionalCanonicalizationTest[
           signedConfigurations, generatingTranspositions] // Timing

         Passed :31

         Failed :0

         Failed generating Transposition sets :{}

Out[89]= {57.9167 Second, Null}
```

Thus, as before, we have verified that transpositional canonicalization works correctly on all possible configurations of up to 6 indices with all possible adjacent transpositional generating sets — but this time allowing fixed elevation indices.

C.2.13 Example: Testing the Conjecture for 6 Indices with Class Extended Indices

Let us now proceed with the first stage of verifying Conjecture 6.13.B. Specifically, we only verify the conjecture for class extended indices. We do not yet include fixed elevations in our test cases. We will assume that either no indices or all indices in a configuration have class extensions. This is valid since associating classes with just some indices is equivalent to treating all the indices without a class as being of the same class.

```
In[91]:= signedConfigurations = addAllClasses @ allConfigurations[6];
         generatingTranspositions = allValidTranspositionSets[6];
```

The kind of configurations we are verifying transpositional canonicalization for are the following.

```
In[93]:= RandomKSubset[signedConfigurations, 5]
```

$$\text{Out[93]}= \Big\{ \{s_{1,2}^{\text{class3}}, s_{2,1}^{\text{class3}}, s_{3,5}^{\text{class1}}, s_{4,6}^{\text{class2}}, s_{5,3}^{\text{class1}}, s_{6,4}^{\text{class2}}\}_+,\ \{s_{1,3}^{\text{class2}}, s_{2,6}^{\text{class1}}, s_{3,1}^{\text{class2}}, s_{4,5}^{\text{class3}}, s_{5,4}^{\text{class3}}, s_{6,2}^{\text{class1}}\}_+,\ \{s_{1,5}^{\text{class2}}, s_{2,4}^{\text{class1}}, s_{3,6}^{\text{class3}}, s_{4,2}^{\text{class1}}, s_{5,1}^{\text{class2}}, s_{6,3}^{\text{class3}}\}_+,\ \{s_{1,6}^{\text{class2}}, s_{2,4}^{\text{class1}}, s_{3,5}^{\text{class1}}, s_{4,2}^{\text{class1}}, s_{5,3}^{\text{class1}}, s_{6,1}^{\text{class2}}\}_+,\ \{s_{1,6}^{\text{class2}}, s_{2,5}^{\text{class1}}, s_{3,4}^{\text{class3}}, s_{4,3}^{\text{class3}}, s_{5,2}^{\text{class1}}, s_{6,1}^{\text{class2}}\}_+ \Big\}$$

We can see that by affixing classes to our indices, we have generated many more configurations than without, but still less than if we included fixed elevation indices in our configurations.

```
In[94]:= signedConfigurations_ten
```

```
Out[94]= 195
```

```
In[95]:= generatingTranspositions_ten
```

```
Out[95]= 31
```

There are 195 configurations and 31 different transpositional generating sets for which we must verify the conjecture.

```
In[96]:= exhaustiveTranspositionalCanonicalizationTest[
             signedConfigurations, generatingTranspositions] // Timing

         Passed :31

         Failed :0

         Failed generating Transposition sets :{}

Out[96]= {31.35 Second, Null}
```

Thus, we have shown the conjecture to be true for all possible adjacent transpositional generating sets and all possible class extended configurations on up to 6 indices.

C.2.14 Example: Testing the Conjecture for 6 Indices with Class Extended Indices and Fixed Elevations

Let us now proceed with fully verifying Conjecture 6.13.B on 6 indices.

```
In[97]:= signedConfigurations = addAllClasses @ allConfigurationsWithElevations[6];
         generatingTranspositions = allValidTranspositionSets[6];
```

We will be verifying transpositional canonicalization for configurations like the following.

```
In[99]:= RandomKSubset[signedConfigurations, 5]
```

$$\text{Out[99]=}\ \Big\{\{s_{1,2}^{\text{class2},\uparrow}, s_{2,1}^{\text{class2},\downarrow}, s_{3,5}^{\text{class2},\downarrow}, s_{4,6}^{\text{class1},\uparrow}, s_{5,3}^{\text{class2},\uparrow}, s_{6,4}^{\text{class1},\downarrow}\}_+,$$
$$\{s_{1,4}^{\text{class3},\uparrow}, s_{2,3}^{\text{class1},\uparrow}, s_{3,2}^{\text{class1},\downarrow}, s_{4,1}^{\text{class3},\downarrow}, s_{5,6}^{\text{class2}}, s_{6,5}^{\text{class2}}\}_+,$$
$$\{s_{1,4}^{\text{class1},\downarrow}, s_{2,5}^{\text{class3}}, s_{3,6}^{\text{class2},\downarrow}, s_{4,1}^{\text{class1},\uparrow}, s_{5,2}^{\text{class3}}, s_{6,3}^{\text{class2},\uparrow}\}_+,$$
$$\{s_{1,5}^{\text{class3},\uparrow}, s_{2,3}^{\text{class2}}, s_{3,2}^{\text{class2}}, s_{4,6}^{\text{class1}}, s_{5,1}^{\text{class3},\downarrow}, s_{6,4}^{\text{class1}}\}_+,$$
$$\{s_{1,5}^{\text{class3}}, s_{2,6}^{\text{class1},\downarrow}, s_{3,4}^{\text{class2},\downarrow}, s_{4,3}^{\text{class2},\uparrow}, s_{5,1}^{\text{class3}}, s_{6,2}^{\text{class1},\uparrow}\}_+\Big\}$$

We can see that by allowing fixed elevations in our class extended indices, we have generated dramatically more configurations than ever before.

```
In[100]:= signedConfigurations_ten
```

```
Out[100]= 5265
```

```
In[101]:= generatingTranspositions_ten
```

```
Out[101]= 31
```

There are 5265 different configurations and 31 different transpositional generating sets for which we must verify the conjecture.

```
In[102]:= exhaustiveTranspositionalCanonicalizationTest[
              signedConfigurations, generatingTranspositions] // Timing

          Passed :31

          Failed :0

          Failed generating Transposition sets :{}

Out[102]= {1956.82 Second, Null}
```

This now verifies Conjecture 6.13.B for all possible adjacent transpositional generating sets and all possible fixed-elevated class-extended configurations on up to 6 indices. As can been seen, the combinatorial explosion is starting to effect us. To test the hypothesis for higher numbers of indices than 6 requires random sampling methods.

C.2.15 Limits to Testing the Conjecture

In this subsection we push our verification to the limit. The results are somewhat uninteresting since we are already extremely confident of the outcome. However, it is nice to explicitly push the algorithm to its limits.

We can verify the basic Conjecture 6.9.A on up to 10 indices.

```
In[103]:= signedConfigurations = allConfigurations[10];
          generatingTranspositions = allValidTranspositionSets[10];

In[105]:= signedConfigurations[ten]

Out[105]= 945

In[106]:= generatingTranspositions[ten]

Out[106]= 511

In[107]:= exhaustiveTranspositionalCanonicalizationTest[
                signedConfigurations, generatingTranspositions] // Timing

          Passed :511

          Failed :0

          Failed generating Transposition sets :{}

Out[107]= {3742.57 Second, Null}
```

This is really the practical limit of our ability to test our basic Conjecture 6.9.A. To test it for 12 indices would take in the vicinity of 33 hours on the author's machine, and for 14 indices it would take approximately 2.3 months.

Once we allow indices with fixed elevations in our configurations, we can only practically perform the above type of computation for up to 8 indices.

```
In[108]:= signedConfigurations = allConfigurationsWithElevations[8];
          generatingTranspositions = allValidTranspositionSets[8];

In[110]:= signedConfigurations[ten]

Out[110]= 8505

In[111]:= generatingTranspositions[ten]

Out[111]= 127

In[112]:= exhaustiveTranspositionalCanonicalizationTest[
                signedConfigurations, generatingTranspositions] // Timing

          Passed :127

          Failed :0

          Failed generating Transposition sets :{}
```

```
Out[112]= {11242.9 Second, Null}
```

Similarly when we allow indices with class extensions, we can practically only perform this computation for up to 8 indices.

```
In[113]:= signedConfigurations = addAllClasses @ allConfigurations[8];
          generatingTranspositions = allValidTranspositionSets[8];

In[115]:= signedConfigurations[[ten]]

Out[115]= 7875

In[116]:= generatingTranspositions[[ten]]

Out[116]= 127

In[117]:= exhaustiveTranspositionalCanonicalizationTest[
                signedConfigurations, generatingTranspositions] // Timing

          Passed :127

          Failed :0

          Failed generating Transposition sets :{}

Out[112]= {11556.3 Second, Null}
```

We should note in conclusion that since intuitively we have enough "room to move" in our configurations, if it were possible to find a contradiction to one of our conjectures, then we would have almost certainly already "found" one.

C.3 Evidence for Direct Reduction Conjecture

C.3.1 Direct Reduction Rule Sets

In this subsection let us briefly give some corroborating evidence to support Conjecture 6.14.A. Let us re-enter the code for transforming a tensor product to its corresponding internal symbol, as well as clearing the cached tensor symbols.

```
In[1]:= toTensorSymbols @ tensors_Plus := toTensorSymbols /@ tensors;
        toTensorSymbols [ nt_ tensorProduct_] :=
           nt toTensorSymbols @ tensorProduct /; FreeQ[nt, Tensor];
        toTensorSymbols @ eqn_Equal := toTensorSymbols /@ eqn;
        toTensorSymbols @ other_ := tensorProductToTensorSymbol @ other;

In[5]:= ClearTensorCaches[]
```

For demonstration purposes, it is convenient to use a more complicated set of governing equations than that used in the previous subsections. Thus, let us obtain the set of governing equations of the following, more complicated, tensor product.

```
In[6]:= equations = toTensorSymbols /@ EquationsOfExpression[R^{α δ β μ}_{;ν} R_{α β δ τ;ε}];
```

There are 49 governing equations for our symmetries.

```
In[7]:= equations_ten

Out[7]= 49
```

Here is a simple function that, given a set of governing equations, creates a corresponding set of reduction rules based on the derived ordering.

```
In[8]:= toReductionRules @ equations_ :=
          Sort @ Flatten[
                With[{variables = Cases[{eqn}, _?tensorProductSymbolQ, {0, ∞}]},
                      Solve[eqn, Last @ sortTensorSymbols @ variables]] &_eqn /@
            equations]

In[9]:= reductionRules = toReductionRules @ equations;
```

These rules look like the following.

```
In[10]:= RandomKSubset[reductionRules, 5]

Out[10]= {TPS`Π16 → TPS`Π21/2, TPS`Π22 → TPS`Π21/2, TPS`Π22 → TPS`Π10 + TPS`Π23,
           TPS`Π30 → -TPS`Π12 + TPS`Π25, TPS`Π4 → TPS`Π7/2}
```

Often, many of the rules have "branchings" in them. That is, there are several different reductions possible from any given initial symbol.

```
In[11]:= reductionRules[[{2, 3}]]

Out[11]= {TPS`Π12 → TPS`Π11/2, TPS`Π12 → TPS`Π14 - TPS`Π9}
```

In fact, we can see the number of "branches" each tensor product symbol has by simply counting the number of times it occurs as the left hand side of one of the reduction rules.

```
In[12]:= Length /@ Split[First /@ reductionRules]

Out[12]= {1, 2, 2, 1, 2, 2, 3, 2, 1, 1, 1, 2,
           3, 1, 1, 1, 2, 2, 1, 2, 2, 2, 3, 4, 2, 1, 1, 1}
```

So for instance, the 11^{th},12^{th}, and 13^{th} rules are all reductions for the same symbol.

```
In[13]:= reductionRules[[{11, 12, 13}]]

Out[13]= {TPS`Π17 → -TPS`Π15 + TPS`Π16,
           TPS`Π17 → TPS`Π18/2, TPS`Π17 → -TPS`Π13 + TPS`Π4}
```

There is, of course, the question of what happens if we use one reduction versus another. Will we obtain the same answer in the end? If we could obtain different answers, then we would be able to find contradictions to our Conjecture 6.14.A. To determine if any such reductions exist, it is convenient to use a data structure for multiple solutions.

C.3.2 Aside: Multiple Solutions

We need to introduce a data structure for "multiple-solutions" objects so we can perform computations which return multiple solutions. We will use the wrapper `MultipleSolutions` for this purpose. Let us denote such objects by the notation $\{\}_M$.

```
In[14]:= Notation[{s___}_M ⇔ MultipleSolutions[s___]]
```

Technical Note: Previously, in §4.6.4 *Dynamic Rules and Assignments III*, the issue of multiple solutions was commented on in a technical note. That discussion is somewhat relevant to the concepts in this subsection. Also, the concept of multiple-solutions as we use them here were first presented in Harris[148].

Intuitively, a multiple-solutions object should behave, unsurprisingly, like a multiple solution. If we perform any arithmetic operations on one of these objects, then that operation should be done on each solution. Also, a multiple-solutions object should be "flat". Formally, `MultipleSolutions` objects have the following properties.

$$\{e_1, e_2\}_M +_{op} expr \longrightarrow \{e_1 +_{op} expr, e_2 +_{op} expr\}_M$$
$$\{e_1, \ldots, \{e_i, \ldots, e_j\}_M, \ldots, e_n\}_M \longrightarrow \{e_1, \ldots, e_i, \ldots, e_j, \ldots, e_n\}_M$$

where $+_{op}$ is restricted to an arithmetic operation and the e_i are expressions. Let us now implement `MultipleSolutions` objects. For the flat-like behavior, we have the following.

```
In[15]:= expr : {___, _MultipleSolutions, ___}_M :=
            Flatten[expr_u, 2, MultipleSolutions]
```

To ensure that we include each sub-solution in our multiple-solutions objects only once, we add the following rule.

```
In[16]:= {solutions__}_M := With[{test = Union @ {solutions}},
            MultipleSolutions @@ test /; test ≠ {solutions}]
```

Finally, since `MultipleSolutions` should distribute over arithmetic type heads, we include the following rules.

```
In[17]:= MultipleSolutions /: expr : (head_?distributeOverQ)[___, {___}_M, ___] :=
            Distribute[expr_u, MultipleSolutions]
```

```
In[18]:= MultipleSolutions @ singleSolution_ = singleSolution;
         distributeOverQ @ symb_Symbol :=
            distributeOverQ@symb = (Listable ∈ Attributes@symb);
         distributeOverQ @ Replace = True;
         distributeOverQ @ ReplaceAll = True;
         distributeOverQ @ ReplaceRepeated = True;
```

We can easily demonstrate the behavior of `MultipleSolutions` objects with the following.

```
In[23]:= {a, b}_M / 2 + {c, d}_M^2
```

$$\text{Out[23]=}\ \left\{\frac{a}{2}+c^2,\ \frac{b}{2}+c^2,\ \frac{a}{2}+d^2,\ \frac{b}{2}+d^2\right\}_M$$

In the remainder of the code in this subsection, it is also convenient to define the following boolean test.

```
In[24]:= isSingleSolutionQ @ any_MultipleSolutions = False ;
         isSingleSolutionQ @ other_ = True;
```

C.3.3 Multiple Reductions

We can now return to our example demonstration that following any reduction strategy allowed by our reducing rules leads to the same final result. Since we now have the use of multiple-solutions objects, we can create multiple reducing rules, that is, rules that will reduce an expression and return all such reductions.

```
In[26]:= multipleReductionRule =
           {term_?tensorProductSymbolQ /; (term ≠ (term /. reductionRules)) :>
                 MultipleSolutions @@ ReplaceList[term, reductionRules]};

In[27]:= reductionRules[[17]] /. multipleReductionRule

Out[27]= -2 TPS`Π1 → -2 TPS`Π1
```

If we then apply our multiple-reduction rule again, we can obtain all possible expressions reachable after two reductions

```
In[28]:= TableForm[List @@ (% /. multipleReductionRule)]

Out[28]//TableForm=
         -2 TPS`Π1
         -2 TPS`Π1
```

If we repeatedly multiply reduce each symbol for which we have a reduction rule, we will obtain the final set of possible answers from using the reduction rules in any order.

```
In[29]:= TPS`Π17 //. multipleReductionRule
```

$$\text{Out[29]=}\ \left\{-\text{TPS`}\Pi6+\text{TPS`}\Pi9,\ \text{TPS`}\Pi1-\text{TPS`}\Pi10+\frac{1}{2}\left(-2\,\text{TPS`}\Pi1+2\,\text{TPS`}\Pi10\right)-\text{TPS`}\Pi6+\text{TP}\right.$$

These reductions are in fact all the same.

```
In[30]:= Expand /@ List @@ %

Out[30]= {-TPS`Π6 + TPS`Π9, -TPS`Π6 + TPS`Π9, -TPS`Π6 + TPS`Π9}
```

This is essentially the method we will use to show that every tensor product symbol in our equations is reduced to a unique answer by our reduction rules. The complete set of tensor product symbols occurring in our governing equations is found by the following.

```
In[31]:= orderedTensorSymbols = Reverse @ sortTensorSymbols @
                  Union @ Cases[{equations}, symb_?tensorProductSymbolQ, {0, ∞}]

Out[31]= {TPS`Π33, TPS`Π28, TPS`Π27, TPS`Π17, TPS`Π18, TPS`Π4,
    TPS`Π7, TPS`Π32, TPS`Π29, TPS`Π31, TPS`Π26, TPS`Π30,
    TPS`Π25, TPS`Π16, TPS`Π15, TPS`Π19, TPS`Π3, TPS`Π5, TPS`Π6,
    TPS`Π22, TPS`Π23, TPS`Π24, TPS`Π21, TPS`Π20, TPS`Π2, TPS`Π1,
    TPS`Π13, TPS`Π8, TPS`Π12, TPS`Π14, TPS`Π9, TPS`Π11, TPS`Π10}

In[32]:= isSingleSolutionQ [ Expand /@ (TPS //. multipleReductionRule)] &TPS /@
    orderedTensorSymbols

Out[32]= {True, True, True, True, True, True, True, True, True, True, True,
    True, True, True, True, True, True, True, True, True, True, True,
    True, True, True, True, True, True, True, True, True, True, True}
```

We can see that no matter how we reduce any of the symbols, we arrive at the same final answer. Thus no matter how we reduce any expression via the reduction rules, we arrive at the same answer. Thus for this example we have confirmed Conjecture 6.14.A.

C.3.4 Reduction Structure

The code developed in the previous subsection brings up a point that we commented on in §6.14.5 *Equational Systems and Gröbner Canonicalization*. After canonicalization, is it possible that a single term will get canonicalized into many terms? Previously, we stated that the answer was "mostly no". Let us examine this question in detail for our example case. The number of terms each symbol is reduced to can be found as follows.

```
In[33]:= numberOfTerms @ any_Plus := Length @ any;
         numberOfTerms @ other_ = 1;

In[35]:= numberOfTerms [ Expand /@ (TPS //. multipleReductionRule)] &TPS /@
    orderedTensorSymbols

Out[35]= {1, 1, 2, 2, 2, 1, 1, 3, 3, 4, 3, 2, 1, 2, 4,
    2, 1, 2, 1, 2, 1, 1, 2, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1}
```

Looking at the above, we can better gauge what is meant by "mostly no". Most of the terms lead to just a single term, but for a small number of terms, it transpires that upon canonicalization, a single term will become an answer with 4 terms. Here is such an example.

```
In[36]:= orderedTensorSymbols[[10]] //. multipleReductionRule

Out[36]= TPS`Π1 + TPS`Π25 - TPS`Π6 + TPS`Π9

In[37]:= orderedTensorSymbols[[10]]

Out[37]= TPS`Π31
```

In fact, this is exactly the same answer as returned by Gröbner canonicalization.

```
In[38]:= Last @ PolynomialReduce[TPS`Π31,
           GroebnerBasis[equations, orderedTensorSymbols], orderedTensorSymbols]

Out[38]= TPS`Π1 + TPS`Π25 - TPS`Π6 + TPS`Π9
```

It should be clear that in the end, all of the symbols are reduced to a fairly select set of symbols which are "smallest".

```
In[39]:= Reverse @ sortTensorSymbols @
           Union @ Cases[{Expand /@ (TPS //. multipleReductionRule)} &TPS /@
                   orderedTensorSymbols, symb_?tensorProductSymbolQ, {0, ∞}]

Out[39]= {TPS`Π25, TPS`Π6, TPS`Π1, TPS`Π9, TPS`Π10}
```

Consequently, any expression made up of terms in the governing equations of our example can be expressed as the linear sum of at most 5 terms. Finally, to close this section, it is interesting to compare the timings of using direct reduction as compared to Gröbner canonicalization.

```
In[40]:= Timing[TPS`Π31 //. toReductionRules @ equations]

Out[40]= {0.283333 Second, TPS`Π1 + TPS`Π25 - TPS`Π6 + TPS`Π9}

In[41]:= Timing[
           orderedTensorSymbols = Reverse @ sortTensorSymbols @ Union @
                   Cases[{equations}, symb_?tensorProductSymbolQ, {0, ∞}];
           Last @ PolynomialReduce[TPS`Π31, GroebnerBasis[equations,
                   orderedTensorSymbols], orderedTensorSymbols]]

Out[41]= {0.633333 Second, TPS`Π1 + TPS`Π25 - TPS`Π6 + TPS`Π9}
```

So we see that using direct reduction leads to a doubling in speed for the linear canonicalization stage.

Finally, it should be noted that we can try even more complex examples to verify our conjecture. However, we cannot use the code given above verbatim. This is because the number of branchings becomes so large that the number of intermediate stages swamp the calculation. We would need to start caching reductions in order to rectify this. The overall situation can be compared to the code for the Fibonacci numbers, defined by: $fib(0) = 1$, $fib(1) = 1$, $fib(n) = fib(n-1) + fib(n-2)$. If we evaluated the Fibonacci numbers without caching, the timings grow exponentially; however with caching, the process becomes linear. The same sort of thing is true for our branchings.

C.4 Canonicalizing in Stages

C.4.1 Symmetry Factorization and Truncated Orderings

We should comment on one final potential optimization we could make to the process of generating configurations. We have not yet taken into account the commutative nature of permutations that operate on different tensors. For instance, it is obvious that if we have a tensor like $\mathbf{A}^{kij}{}_{k}\mathbf{S}^{a}{}_{ij}$ then any permutation acting on S commutes with any permutation acting on A. We can exploit this commutativity of generators to factor the generation of configurations into stages. Once we can generate all configurations in stages, then at each stage we can eliminate all configurations that we know will lead to larger configurations. Such an optimization has the potential to lead to the generation of far fewer configurations in our search for the canonical configuration.

Let us first motivate the algorithm by presenting a concrete example of when we can prune configurations in stages. Consider the tensor $\mathbf{R}^{abcd}{}_{cab}\mathbf{R}^{e}{}_{edij}\mathbf{S}$. Assume that after generating all possible configurations using only the symmetries of R, we have obtained just the following two configurations:

$$\mathbf{R}^{abcd}{}_{cab}\mathbf{R}^{e}{}_{edij}\mathbf{S} \text{ and } \mathbf{R}^{abcd}{}_{abc}\mathbf{R}^{e}{}_{edij}\mathbf{S}$$

It should be apparent from considering our ordering on configurations, that no matter what permutations involving S are applied to the first tensor configuration, the resulting configuration will always be larger than the result of applying any permutations to the second configuration. This is because the free indices are in the same positions and the summed indices are in the same shape, and therefore the order is determined solely on the relative values of the summed indices.

We can make the above clearer by setting all indices involved in S, and any tensor factors further to the right, to the symbol ■. This allows us to compare the two configurations on just the indices that have been so far fixed.

$$\mathbf{R}^{abc■}{}_{cab}\mathbf{R}^{■}{}_{■■■■}\mathbf{S} \text{ vs. } \mathbf{R}^{abc■}{}_{abc}\mathbf{R}^{■}{}_{■■■■}\mathbf{S}$$

With this new pseudo-tensorial form, it should be clearer that no matter how the permutations on S move the indices in S, there is no way that the second configuration can be larger than the first with the same permutations on S, since the shape of the two configurations will always be the same. In general, applying permutations to one particular tensor factor *cannot* affect the

shape of *other* tensor factors in the tensor product. However, they may change the ordering of the indices in the other factors.

We can confirm that once these pseudo-forms are translated to configurations, the first configuration is larger than the second.

```
In[1]:= ⟨R^abc■ R^■_cab S_■■■■⟩_c ≻_c ⟨R^abc■ R^■_abc S_■■■■⟩_c

Out[1]= True
```

Definition 13.4.A: A set of generators S is *factorizable* if S can be factored (or partitioned) into disjoint subsets S_i such that generators from distinct factors commute. That is, $S = \bigcup_i S_i$, such that $\forall\ s \in S_i,\ t \in S_j$ then $s \cdot t = t \cdot s$ if $i \neq j$.

The nature of the permutation generators for the various tensor factors ensure that the total permutation group of a tensor product is factorizable into an internal direct product of the permutation groups for each individual tensor *cluster*. Clusters of tensors are tensor factors that are linked by degeneracy symmetries. (The issues surrounding degeneracy symmetries and factorization were commented on in §6.8.2 *Examining the Classification of Symmetries via GAP*.) The permutation group for a tensor product is always factorizable in terms of the subgroups corresponding to the various tensor product factors. This is because (*i*) these subgroups have just the identity in common, (*ii*) elements from these different subgroups commute with each other, and (*iii*) the only subgroup of the overall permutation group containing all of the above mentioned subgroups is the overall permutation group itself — see Fraleigh[114], Rose[276], or MacLane[216].

Theorem 13.4.A: Given a configuration $a \in C$ and some permutation $\rho \in \mathcal{G}$ that does not move the indices in the part of a corresponding to a tensor factor T, then the shape of the indices in the T part of $\rho *_c a$ are the same as the shape of the indices in the T part of a.

Proof: Obvious. ■

The purpose of factorization is that it allows us to remove those configurations at each stage which we know will lead to larger configurations. Thus, from an initial configuration, we generate all configurations due to the symmetries of the first tensor. From this set we eliminate any configurations that we know will lead to non-minimal configurations. Then, with this reduced set, we generate all configurations reachable by the second set of symmetries. We then eliminate the configurations that will lead to non-minimal configurations, and proceed to the third set of symmetries, and so on.

Now that we know why we want to generate configurations in stages, let us proceed to do so. First, we need to be able to partially compare configurations on just the indices that have been permuted so far.

```
In[2]:= Notation[a_ ≲^n_cp b_ ⟺ truncatedSignedConfigurationOrder[a_, b_, n_]]
```

```
In[3]:= sc1_signedConfiguration ≲^n_cp sc2_signedConfiguration :=
          truncateConfiguration[sc1, n] ≲_c truncateConfiguration[sc2, n]
```

To truncate a configuration, we just take only the part that has already been involved in the permutations. We transform all summed indices in this truncated part which have a link outside the part to a uniform $s_{i,j}$.

```
In[4]:= truncateConfiguration[config_sign_, n_] :=
          Take[config, n] /. s_i_,j_ /; j > n ∨ i > n → s_i,n sign
```

We can see that this truncated ordering is the standard ordering on the elements so far permuted.

```
In[5]:= ⟨R^abcd R^e_cab S_edij⟩_c ≲^8_cp ⟨R^abcd R^e_abc S_edij⟩_c

Out[5]= -1
```

However, the indices after the cutoff point are not distinguished by the partial comparison operator. For example, in the following the indices e and d in the tensor S are not as ordered in the second configuration as they are in the first.

```
In[6]:= ⟨R^abcd R^e_abc S_deij⟩_c ≲^8_cp ⟨R^abcd R^e_abc S_edij⟩_c

Out[6]= 1
```

Despite being different tensors, our truncated ordering operator says that they are the same configurations according to their truncated parts.

C.4.2 Canonicalizing in Stages

Let us proceed to apply the ideas of the previous subsection and create a simplistic version of our algorithm to investigate how "canonicalizing in stages" works in practice.

To the above end, let us create a simple function that eliminates all configurations which are "larger" than the "smallest" configurations in a given set. The algorithm collates the configurations which are the same under the partial order by scanning over a list of configurations. If a configuration of the same "size" is encountered, it is added to the collated configurations. If a configuration that is smaller is encountered, the collection is reset to just this configuration. If a larger configuration is found, it is discarded. In this way we obtain the subset of smallest configurations under the partial order. This process is linear in the size of the configuration set.

```
In[7]:= removeLargerConfigurations[configurations_List, n_Integer?Positive] :=
          Block[{collection = {}, smallest = First @ configurations},
            handleConfiguration[sc, sc ≲^n_cp smallest] &_sc /@ configurations;
            collection]
```

```
In[8]:= handleConfiguration[sc_signedConfiguration, 0] := AppendTo[collection, sc]
        handleConfiguration[sc_signedConfiguration, 1] :=
            (smallest = sc; collection = {sc})
```

Let us examine how this algorithm works in practice. First let us use the following as our test configuration.

```
In[10]:= signedConfig = ⟨R^{cmdn} R_{c ma}^{a} S_{n bd}^{b}⟩_c ;
```

This tensor product has the following symmetries.

```
In[11]:= ⟨R^{cmdn} R_{c ma}^{a} S_{n bd}^{b}⟩_{S_T, S_X, S_⊗}

Out[11]= {{(1 ↔ 2)_, (3 ↔ 4)_, (5 ↔ 6)_, (7 ↔ 8)_, (9 ↔ 10)_+, (10 ↔ 11)_+,
            (11 ↔ 12)_+}, {{3, 4, 1, 2, 5, 6, 7, 8, 9, 10, 11, 12}_+,
            {1, 2, 3, 4, 7, 8, 5, 6, 9, 10, 11, 12}_+},
          {{5, 6, 7, 8, 1, 2, 3, 4, 9, 10, 11, 12}_+}}
```

So let us manually split this set of symmetries into the symmetries for the cluster RxR and the tensor factor S.

```
In[12]:= Symbolize[σ_RxR]; Symbolize[σ_S];

In[13]:= σ_RxR = {(1 ↔ 2)_, {1, 2, 3, 4, 7, 8, 5, 6, 9, 10, 11, 12}_+,
                  {3, 4, 1, 2, 5, 6, 7, 8, 9, 10, 11, 12}_+};
         σ_S = {(9 ↔ 10)_+, (10 ↔ 11)_+, (11 ↔ 12)_+};
```

We must also slightly update the routine for the generation of configurations from §6.5.2 *The Algorithm for Generating Configurations*. This slight modification just allows us to start with a set of configurations rather than a single seed configuration.

```
In[15]:= generateConfigurations[
             {seedConfigurations__signedConfiguration}, S_List] :=
           Block[{F = {}, N = {seedConfigurations}},
             While[N ≠ {},
               F = F ⋃ N;
               N = (S *_c N) \ F];
             F]
```

Now, starting with our signed configuration, by applying the symmetries for the RxR cluster, we obtain 16 different configurations.

```
In[16]:= generateConfigurations[signedConfig, σ_RxR]_ten

Out[16]= 16
```

However, if we eliminate from this collection all configurations which we conclusively know will lead to non-minimal configurations, we obtain only 4 configurations.

```
In[17]:= partiallyGeneratedConfigurations = removeLargerConfigurations[
             generateConfigurations[signedConfig, σ_RxR], 8];
         partiallyGeneratedConfigurations_ten
```

```
Out[18]= 2
```

If we continue and generate the signed configurations reachable by the symmetries of our second tensor factor, we obtain 24 configurations.

```
In[19]:= generateConfigurations[partiallyGeneratedConfigurations, σS]//Length

Out[19]= 24
```

Thus we have only generated 16+24=40 configurations, compared to 96 configurations if we generated them in the conventional way.

```
In[20]:= generateConfigurations[signedConfig, σR×R ⋃ σS]//Length

Out[20]= 96
```

It is instructive to compare the timings taken by these two approaches. Conventionally, to generate all configurations and find the minimum one for our tensor product would take the following length of time.

```
In[21]:= Min_c @ generateConfigurations[signedConfig, σR×R ⋃ σS] // Timing_50

Out[21]= {0.554333 Second,
          {sg_{1,5}, sg_{2,7}, sg_{3,10}, sg_{4,11}, sg_{5,1}, sg_{6,8}, sg_{7,2}, sg_{8,6}, sg_{9,12}, sg_{10,3}, sg_{11}
          }
```

Compare this to the same procedure in the truncated approach.

```
In[22]:= Min_c @ generateConfigurations[removeLargerConfigurations[
                  generateConfigurations[signedConfig, σR×R], 8], σS] // Timing_50

Out[22]= {0.173 Second,
          {sg_{1,5}, sg_{2,7}, sg_{3,10}, sg_{4,11}, sg_{5,1}, sg_{6,8}, sg_{7,2}, sg_{8,6}, sg_{9,12}, sg_{10,3}, sg_{11}
          }
```

In this case, we can see that the truncated approach is around three times faster. The next subsection examines how these results change once we include transpositional canonicalization

C.4.3 Canonicalizing in Stages: Incorporating Transpositional Canonicalization

This subsection examines how the results of the above subsection change once we include transpositional canonicalization into the algorithm for canonicalizing in stages. The test example in the previous subsection included tensor factors with degeneracy, as well as having a factor that was completely transpositional in nature. Let us therefore switch to a more complicated and suitable example for examining canonicalizing in stages while including transpositional canonicalization. Here is the tensor product we will use.

```
In[23]:= signedConfig = ⟨C^{a c}_{b a} K^{m n}_{d n} U^{i j}_{c i} W^{b d}_{j m}⟩_c ;
```

Throughout this subsection we will usually show only the indices of the signed configuration and omit the sign, in order to stop the display of the signed configurations running off the printed page.

```
In[24]:= signedConfig[[2]]

Out[24]= {sg1,3, sg2,13, sg3,1, sg4,11, sg5,16, sg6,15, sg7,8, sg8,7,
          sg9,12, sg10,14, sg11,4, sg12,9, sg13,2, sg14,10, sg15,6, sg16,5}
```

Let us also give Riemann-like symmetries to each of these tensor factors. (We have chosen different names since if they all had the same name, then there would be degeneracy symmetries between them, hence they would only form a single tensor cluster, hence there would be no difference between generating in stages and the conventional approach.)

```
In[25]:= DeclareSymmetries[U, 4, {(1 ↔ 2)_, (3 ↔ 4)_, {3, 4, 1, 2}_+}];
         DeclareSymmetries[C, 4, {(1 ↔ 2)_+, (3 ↔ 4)_+, {3, 4, 1, 2}_+}];
         DeclareSymmetries[K, 4, {(1 ↔ 2)_, (3 ↔ 4)_, {3, 4, 1, 2}_+}];
         DeclareSymmetries[W, 4, {(1 ↔ 2)_, (3 ↔ 4)_, {3, 4, 1, 2}_+}];
```

As before, we must slightly update the generation of configurations via closure and transpositional canonicalization. Again as before, this slight change allows us to start with a set of configurations rather than a single seed configuration.

```
In[29]:= generateConfigurations_T_List[
             {seedConfigurations__signedConfiguration}, R_List] :=
           Block[{F = {}, N = Θ_T[{seedConfigurations}]},
             While[N ≠ {},
               F = F ⋃ N;
               N = Θ_T[R *_c N] \ F;];
             F]
```

The tensor product we have chosen has the following symmetries.

```
In[30]:= ⟨C^a_ba^c_d K^mn_n U^ij_ci W^b_jm^d⟩_S_T,X

Out[30]= {{(1 ↔ 2)_+, (3 ↔ 4)_+, (5 ↔ 6)_-, (7 ↔ 8)_-,
           (9 ↔ 10)_-, (11 ↔ 12)_-, (13 ↔ 14)_-, (15 ↔ 16)_-},
          {{3, 4, 1, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}_+,
           {1, 2, 3, 4, 7, 8, 5, 6, 9, 10, 11, 12, 13, 14, 15, 16}_+,
           {1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 9, 10, 13, 14, 15, 16}_+,
           {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 15, 16, 13, 14}_+}}
```

Thus, let us again split up this generating set into its factorizable parts.

```
In[31]:= Symbolize[σ_C]; Symbolize[σ_K]; Symbolize[σ_U]; Symbolize[σ_W];

In[32]:= Symbolize[T_C]; Symbolize[T_K]; Symbolize[T_U]; Symbolize[T_W];

In[33]:= T_C = {(1 ↔ 2)_+, (3 ↔ 4)_+};
         T_K = {(5 ↔ 6)_-, (7 ↔ 8)_-};
         T_U = {(9 ↔ 10)_-, (11 ↔ 12)_-};
         T_W = {(13 ↔ 14)_-, (15 ↔ 16)_-};
```

```
In[37]:= σC = {{3, 4, 1, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}+};
         σK = {{1, 2, 3, 4, 7, 8, 5, 6, 9, 10, 11, 12, 13, 14, 15, 16}+};
         σU = {{1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 9, 10, 13, 14, 15, 16}+};
         σW = {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 15, 16, 13, 14}+};
```

In stages, we would canonicalize our configuration as follows.

```
In[41]:= (partial1 = removeLargerConfigurations [
                       generateConfigurations_Tc [signedConfig, σC], 4];
          partial2 = removeLargerConfigurations [
                       generateConfigurations_TK [partial1, σK], 8];
          partial3 = removeLargerConfigurations [
                       generateConfigurations_TU [partial2, σU], 12];
          partial4 = Min_c @ generateConfigurations_TW [partial3, σW]) [[2]] // Timing_100

Out[41]= {0.055 Second, {sg1,4, sg2,11, sg3,13, sg4,1, sg5,15, sg6,16, sg7,8, sg8,7,
           sg9,12, sg10,14, sg11,2, sg12,9, sg13,3, sg14,10, sg15,5, sg16,6}}
```

Conventionally, when we use transpositional canonicalization, we would proceed as follows.

```
In[42]:= S_T = {(1 ↔ 2)+, (3 ↔ 4)+, (5 ↔ 6)-, (7 ↔ 8)-,
                (9 ↔ 10)-, (11 ↔ 12)-, (13 ↔ 14)-, (15 ↔ 16)-};

In[43]:= S_X = {{3, 4, 1, 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}+,
                {1, 2, 3, 4, 7, 8, 5, 6, 9, 10, 11, 12, 13, 14, 15, 16}+,
                {1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 9, 10, 13, 14, 15, 16}+,
                {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 15, 16, 13, 14}+};

In[44]:= Min_c [ generateConfigurations_S_T [signedConfig, S_X]] [[2]] // Timing_50

Out[44]= {0.330667 Second, {sg1,4, sg2,11, sg3,13, sg4,1, sg5,15, sg6,16, sg7,8, sg8,7,
           sg9,12, sg10,14, sg11,2, sg12,9, sg13,3, sg14,10, sg15,5, sg16,6}}
```

We see that our answers agree and that again canonicalizing in stages is around 3 times faster than conventional canonicalizing (both times including transpositional canonicalization). Even in this extreme test case, when canonicalizing in stages has the best "working conditions" that it is typically likely to have in practice, we have only obtained a speed increase of around a factor of 3.

In practice, it is likely that several of the tensor factors will be degenerate, thus obviating that portion of the "speed increase". Also, it is not entirely guaranteed that canonicalizing in stages will be faster. This is because removing the larger configurations takes a noticeable length of time in the algorithm, and it may well be the case that very few are removed if the truncated ordering does not distinguish between them. Let us briefly show this, by considering the following contrived example.

```
In[45]:= signedConfig = ⟨C^abcd K^ijmn U_aicm W_bjdn⟩_c;
         signedConfig[[2]]

Out[46]= {sg1,9, sg2,13, sg3,11, sg4,15, sg5,10, sg6,14, sg7,12, sg8,16,
           sg9,1, sg10,5, sg11,3, sg12,7, sg13,2, sg14,6, sg15,4, sg16,8}
```

```
In[47]:= (partial1 = removeLargerConfigurations [
                        generateConfigurations_{T_C} [signedConfig, σ_C], 4];
         partial2 = removeLargerConfigurations [
                        generateConfigurations_{T_K} [partial1, σ_K], 8];
         partial3 = removeLargerConfigurations [
                        generateConfigurations_{T_U} [partial2, σ_U], 12];
         partial4 = Min_c @ generateConfigurations_{T_W} [partial3, σ_W]) [[2]] // Timing_100

Out[47]= {0.0501667 Second, {sg_{1,9}, sg_{2,13}, sg_{3,11}, sg_{4,15}, sg_{5,10}, sg_{6,14}, sg_{7,12},
           sg_{8,16}, sg_{9,1}, sg_{10,5}, sg_{11,3}, sg_{12,7}, sg_{13,2}, sg_{14,6}, sg_{15,4}, sg_{16,8}}}
```

Comparing the above timings to those obtained when using the conventional approach, we see that our extra efforts have reduced the computation time only marginally.

```
In[48]:= Min_c [generateConfigurations_{S_T} [signedConfig, S_X]] [[2]] // Timing_50

Out[48]= {0.137 Second, {sg_{1,9}, sg_{2,13}, sg_{3,11}, sg_{4,15}, sg_{5,10}, sg_{6,14}, sg_{7,12}, sg_{8,16},
           sg_{9,1}, sg_{10,5}, sg_{11,3}, sg_{12,7}, sg_{13,2}, sg_{14,6}, sg_{15,4}, sg_{16,8}}}
```

In any case, once we canonicalize the free indices, the number of symmetries usually decreases. In addition to all of the above, encoding and reconstituting a tensor product takes up a significant fraction of the total time to canonicalize a tensor product, so any efficiency gains in generating configurations will be somewhat diluted. Moreover, there are still some questions that must be readdressed. For instance, the question of the determination of a tensor being identically zero must be considered again, since we no longer generate $\mathcal{G}/\mathcal{T} *_c t$, but only a subset of this set.

In summary, it appears that it is not absolutely critical that we add the optimization of canonicalizing in stages to our overall algorithm. However, in most practical cases, some degree of improvement should be achievable. Further work needs to be done on this subject.

Appendix D
Tensor Simplification Code

This appendix contains the complete code and ancillary routines for the tensor simplification algorithms presented within the main thesis. It was written for *Mathematica* 3.0 and *Mathematica* 4.0. Ordinarily such code would not be included directly in a thesis; however, it would be hard to verify the correctness of the canonicalization algorithms without it. For comparison, the source code for the *Assign* package, the *Notation* package, and the *Prototypes* package have all been omitted from the printed form of this thesis.

D.1 Set Up Package Beginnings

D.1.1 Private Preamble

- Wipe Old Package if Present

```
Block[{Message},
   Unprotect /@ (Names @ "Tensors`Private`*" ⋃ Names @ "Tensors`*");
   ClearAll /@ (Names @ "Tensors`Private`*" ⋃ Names @ "Tensors`*");]
```

- Set Input Notebook

We make a copy of the input notebook since it takes a little while to load the package, and the user might select a new notebook in the intervening period while the package is loading.

```
Tensors`Private`inputNotebook = InputNotebook[];
```

- Set $contextSearchPath

We make a copy of the `ContextPath` in order that we may search it for symbols which the *Tensors* package must own.

```
Tensors`Private`$contextSearchPath = $ContextPath

{"Global`", "System`"}
```

D.1.2 BeginPackage

```
BeginPackage["Tensors`",
   "Utilities`Notation`", "Utilities`FilterOptions`"];
```

D.1.3 Private Utilities

```
Begin @ "`Private`";
```

- Timing functionality

```
Notation[
   Timing_n_ ⟹ Function[{expr}, AverageTiming[n_, expr], HoldAll]];
SetAttributes[AverageTiming, HoldAll];
AverageTiming[n_Integer /; n > 1, expr_] := MapAt[# / n &,
      Timing[Do[Unevaluated @ expr;, {n - 1}]; Unevaluated @ expr], 1];
```

- friendlyOff / friendlyOn

`friendlyOff` will turn off a message. `friendlyOn` will turn that message on only if it was on before the `friendlyOff`.

```
SetAttributes[
   {friendlyOff, friendlyOn, messageStatus, wipe}, HoldAll];
messageStatus @ MessageName[func_, mesg_String] :=
   If[ Head [ MessageName[func, mesg] /. Messages @ func] =!= $Off,
      $On, $Off, $Off];
friendlyOff @ mesg_MessageName := (mesgWasOn @ Hold @ mesg =
         (messageStatus @ mesg =!= $Off); Off @ mesg;);
friendlyOn @ mesg_MessageName := If[mesgWasOn @ Hold @ mesg,
      On @ mesg; mesgWasOn @ Hold @ mesg = False;];

friendlyOff @ General::spell1;
```

- Setup basic error handling routines

Some basic functions which the error handling routines uses.
heldLength just gives the length of an expression without evaluating anything.
isNot will return true for anything that does not match the pattern.
headIsNot will return true for anything whose head does not match the pattern.

```
SetAttributes[{heldLength, headIsNot, isNot}, HoldAll]
heldLength @ expr_  := Length @ Unevaluated @ expr;
isNot @ pattern_  :=
   Function[testHead, ¬ MatchQ[Unevaluated @ testHead, pattern], HoldAll];
headIsNot @ pattern_ := Function[testHead,
      ¬ MatchQ[Head @ Unevaluated @ testHead, pattern], HoldAll];

General::optval = "Option of the form `4` → `5` or `4` :→ `5`
         expected (instead of `1`) at position `2` in `3`.";
```

■ End preamble

```
End[];
```

D.1.4 Usage Statements

```
R::usage = "R represents the Riemann or Ricci
         tensor depending on the number of indices.";

S::usage = "S represents the totally symmetric tensor.";

A::usage = "A represents the totally anti-symmetric tensor.";

AddAuxiliaryEquations::usage =
   "AddAuxiliaryEquations[{equations}] declares that the equations
         equations are to be used in conjunction with canonicalizing
         when linear symmetries are being included.";

BaseIndices::usage =
   "BaseIndices[class] returns all the base or root indices
         which belong to the index class class. ";

BasicCanonicalize::usage =
   "BasicCanonicalize[expr] transforms each tensor occurring in
         expr into canonical form according to the symmetries
         previously declared. It works on a more simplistic and
         primitive algorithm than Canonicalize, consequently
         it is slower, however it is included since users
         might want to verify for themselves that the fully
         optimized algorithm returns identical results.";

Canonicalize::usage =
   "Canonicalize[expr] transforms each tensor occurring
         in expr into canonical form according to
         the symmetries previously declared.";

CanonicalizeBrief::usage =
   "CanonicalizeBrief[expr] transforms each tensor occurring in
         expr into a semi-canonical form according using only
         the transpositional symmetries previously declared.";

ClassOfIndex::usage =
   "ClassOfIndex[index] returns the class which index belongs to.";
```

```
ClearTensorCaches::usage =
   "ClearTensorCaches[] will clear all tensorial equation
         sets that have been cached as a by-product of
         canonicalizations. This function is probably only useful
         to regain memory during extremely large calculations.";

CoordinateQ::usage =
   "CoordinateQ[coord] determines if coord is a coordinate.";

DeclareCoordinates::usage =
   "DeclareCoordinates[{c1,c2,…,cn}] declares that
         the ci are to be treated as valid coordinates.
         Note: a coordinate cannot be an index";

DeclareIndexClass::usage =
   "DeclareIndexClass[class,{i1,…,in}] declares that the indices i1
         through in are the base indices for the index class class.";

DeclareCoordinateClass::usage =
   "DeclareIndexClass[class,{c1,c2,…,cm}] declares
         that the coordinates of each index in the index
         class class range over the coordinates c1, …, cm";

DeclareTensorialMultiplicativeHead::usage =
   "DeclareTensorialMultiplicativeHead[head] declares
         that the symbol head will be treated as a
         multiplicative head to be canonicalized over.";

Commutative::usage =
   "Commutative is an option for various functions which
         states the commutativity of specific operations.";

DeclareSymmetries::usage =
   "DeclareSymmetries[name,num,symmetries] declares that the tensor
         named name, having num indices has the symmetries
         symmetries. DeclareSymmetries[name,{n,ncd,npd},symmetries]
         declares that the tensor named name, having n
         standard indices, ncd covariant derivatives, and npd
         partial derivatives has the symmetries symmetries.";

EquationsOfExpression::usage =
   "EquationsOfExpression[expr] returns all tensorial
         equations arising from the linear symmetries of
         the tensors in the tensorial expression expr.";

SpaceTimeIndices::usage =
   "SpaceTimeIndices is the index class for space
         time indices (four vector indices).";

GeneralIndices::usage =
   "GeneralIndices is the index class for general indices.";

High::usage = "High[index], or index+, represents a high
         or contravariant index in a tensorial object.";

IndeterminateIndexClass::usage =
   "IndeterminateIndexClass is the index class for coordinates and
         other things that should not be treated as dummy indices.";
```

```
IndexQ::usage = "IndexQ[index] determines whether index is an index.";

IndexClassInclusion::usage =
   "IndexClassInclusion is a boolean option for Canonicalize
        and related functions. If set to True then the
        class of each index will be internally included
        in the canonicalization algorithm. It is used in
        cases when dummy indices of different types are
        used in the same tensor.The default is True.";

LinearSymmetryMethod::usage =
   "LinearSymmetryMethod is an option to Canonicalize. Their
        are three possible values it can take: GröbnerBases,
        DirectReduction, and None. The default is GröbnerBases.
        If the option is set to None then linear symmetries will
        not be included in the canonicalization of expressions.";

GröbnerBases::usage =
   "GröbnerBases is one of the option values for the option
        LinearSymmetryMethod for the function Canonicalize.
        If this value is chosen then finding a canonical form
        for a tensorial expressions, with respect to a set of
        polynomial tensor equations, is done via GröbnerBases.";

DirectReduction::usage =
   "DirectReduction is one of the option values for the option
        LinearSymmetryMethod for the function Canonicalize.
        If this value is chosen, then finding a canonical
        form for a tensorial expression with respect to a set
        of polynomial tensor equations is done via a direct
        reduction method. This method is faster than others
        but may not give optimal results. (The reductions
        it does give are still guaranteed to be correct.)";

LinearSymmetry::usage =
   " LinearSymmetry[permutations] or \!\(⟨permutations⟩\_\(Σ== \
        0\)\) represents a linear symmetry, where each
        permutation in permutations is a SignedPermutation.";

Low::usage = "Low[index], or index⁻, represents a
        low or covariant index in a tensorial object.";

MetricLocallyFlat::usage =
   "MetricLocallyFlat is a boolean option for Canonicalize. If true
        then the elevations of pairs of dummy indices can be
        freely switched irrespective of partial derivatives.  ";

OptimizedCanonicalize::usage =
   "OptimizedCanonicalize[expr] transforms each tensor occurring in
        expr into canonical form according to the symmetries
        previously declared. It is a simplified version
        of the full canonicalize function. It has largely
        been included for pedagogical purposes. It ignores
        linear symmetries and does not cache any values.";

ReturnOptimizedEquations::usage =
   "ReturnOptimizedEquations is a boolean option
        for EquationsOfExpression. If true then the
```

```
        equations returned by EquationsOfExpression
        will be transformed into a Gröbner basis form
        before being returned. The default is False.";

SignedPermutation::usage =
   "SignedPermutation[sign, {permutation}] or {permutation}± represent a
        permutation together with an overall + or - sign.";

SignedTransposition::usage =
   "SignedTransposition[sign, {i, j}] or (i↔j)± represents
        a transposition with an overall + or - sign.";

StandardSymmetries::usage =
   "StandardSymmetries[tensorProduct] or ⟨tensorProduct⟩S returns
        the standard symmetries for the tensor product
        tensorProduct. The standard symmetries consist of
        the transpositional symmetries, the complex
        symmetries, and the degeneracy symmetries.";

Symmetries::usage =
   "Symmetries[tensorProduct, \"sym1\", \"sym2\",...] or \!\(⟨tensorProduct
        \()\_\(S\_\(sym\_1, \ \ sym\_2, \ \ ...\)\)\)\) returns
        the symmetries for the tensor product tensorProduct
        corresponding to the symi. The symi can be any of
        𝒯, 𝒳, 𝒟, ℒ, 𝒯★. These respectively represent the
        labels for transpositional, complex, degeneracy,
        linear, and transpositional closure symmetries.";

Tensor::usage = "Tensor[𝒯, {i1,..., in}] represents
        the tensorial object 𝒯 with indices i1 through
        in. These indices should be either High or Low.";

UsedIndices::usage = "UsedIndices[expr] returns the
        list of indices used in the expression expr.";

ValidIndexClasses::usage =
   "ValidIndexClasses returns all the valid index classes in use.";

ValidIndexClassQ::usage = "ValidIndexClassQ[class]
        determines whether class is a valid index class.";

ClassOfIndex::usage =
   "ClassOfIndex[index] return the class which index belongs too ";

Dummify::usage = "Dummify[expr] reindexes all dummy
        indices in the expression expr, into unique dummy
        indices which have not previously been used.";

DummyIndices::usage = "DummyIndices[expr] returns a list of
        dummy indices occurring in the expression expr.";

ExpandContraction::usage =
   "ExpandContraction[expr] will expand out any contracted
        dummy indices into a sum of terms where the dummy
        indices range over their allowed coordinates.";

ReIndex::usage = "ReIndex[expr] reindexes all
        dummy indices occurring in the expression expr.";
```

```
UsedIndices::usage = "UsedIndices[expr] returns a
        list of all indices used in the expression expr.";
```

D.1.5 Begin Private

```
Begin @ "`Private`";
```

The following avoids spelling errors being displayed in the package.

```
friendlyOff @ General::spell1;
```

```
env ; tag ; lhs ; rhs;
```

D.1.6 Private Usage Statements

D.2 Set Up Notations

D.2.1 Set Up Package Notations

These notations are used during the creation of the "compiled" package, yet they are not present when the package loads.

```
InfixNotation[+{}, Join]
```

```
Notation[patt_hp ⟺ HoldPattern[patt_]]
```

```
Notation[expr_hc ⟺ HoldComplete[expr_]]
```

```
Notation[expr_hf ⟹ HoldForm[expr_]]
```

```
Notation[expr_u ⟺ Unevaluated[expr_]]
```

```
Notation[body_&λ_ ⟺ Function[{λ__}, body_]]
```

```
Notation[body_&λ__attr__ ⟺ Function[{λ__}, body_, {attr__}]]
```

```
InfixNotation[≠, UnsameQ]
```

```
InfixNotation[≡, SameQ]
```

```
Notation[elem_ ∈? expr_ ⟺ MemberQ[expr_, elem_]]
```

```
Notation[elem_ ∉? expr_ ⟺ FreeQ[expr_, elem_, {1}]]

Notation[patt_ ?~ test_ ⟹ patt_? (! test_[#] &)]

Notation[patt_ ?! test_ ⟹ patt_? (! test_[#] &)]

Notation[ℓ_len ⟺ Length[ℓ_]]

Notation[a_ ≼lex b_ ⟺ OrderedQ[{a_, b_}]]

Notation[a_ ⋠lex b_ ⟹ Not[OrderedQ[{a_, b_}]]]

Notation[a_ ≺lex b_ ⟹ Not[OrderedQ[{b_, a_}]]]

InfixNotation[\, Complement]

Notation[a_ \∉ b_ ⟺ unorderedComplement[a_, b_]]

list_List \∉ out_List := DeleteCases[list, Alternatives @@ out]

Off[General::"spell1"]

Symbolize[S𝒯]; Symbolize[S𝒳]; Symbolize[S𝒟]; Symbolize[S𝓛];
Symbolize[S′𝒯]; Symbolize[S′𝒳]; Symbolize[S′𝒟]; Symbolize[S′𝓛];
Symbolize[(S𝒯)■]; Symbolize[𝒯■];
```

D.2.2 Implementation of Tensor Notation

```
Notation[a_⁺ ⟺ High[a_]]
Notation[a_⁻ ⟺ Low[a_]]
```

First let us handle the extremely simple default case. This occurs when the tensor has no indices, that is, a scalar.

```
Notation[Γ_ ⟹ Tensor[Γ_, {}]]

Notation[makeTensorName[Γ_] ⟸ Tensor[Γ_, {}]]
```

This notation differs slightly from the one described in the discussion section. The reason is that it is nice to parenthesize the "name" of the tensor if it is not a single symbol.

```
Notation[makeTensorName[Γ_] makeGridBox[indices_] ⟸
      Tensor[Γ_, indices_?validIndicesQ]]

makeTensorName @ name_Symbol := MakeBoxes @ name;
makeTensorName @ name_Tensor :=
   RowBox[{"(", MakeBoxes @ name, ")"}];
makeTensorName @ name_ := Parenthesize[name, StandardForm, Power];

validIndicesQ @
      {indices____u ? (MatchQ[#u, High[_] | Low[_]] &)} = True;
validIndicesQ @ _ = True;

makeGridBox @ indices_List :=
   GridBox @ Transpose[makeStringIndexPair /@ indicesu];
```

```
makeStringIndexPair @ High @ index_ := {MakeBoxes @ index, ""};
makeStringIndexPair @ Low @ index_ := {"", MakeBoxes @ index};
makeStringIndexPair @ Low[";"] = {"", ";"};
makeStringIndexPair @ Low[","] = {"", ","};
makeStringIndexPair @ index_ :=
    {"", SuperscriptBox[MakeBoxes @ index,
        StyleBox["*", "TensorStarStyle"]]}

SetAttributes[{validIndicesQ, makeGridBox,
    makeStringIndexPair, makeTensorName}, HoldAll];

Notation[
  Γ_ stringIndices_?validStringIndicesQ ⟹ Tensor[Γ_, makeIndices[stringIndices_]]]
```

These notations are actually different; the second one has the adjustment box and style box striped from it. This allows the parsing of nested tensors. It also circumvents possible style striping in different *Mathematica* front ends.

```
Notation[Γ_ stringIndices_?validStringIndicesQ ⟹
    Tensor[StripBoxes[Γ_], makeIndices[stringIndices_]]]

Notation[Γ_ stringIndices_?validStringIndicesQ ⟹
    Tensor[StripBoxes[Γ_], makeIndices[stringIndices_]]]

validStringIndicesQ @ GridBox[ indices_List, ___] :=
  And @@ (validStringIndexPairQ /@ Transpose @ indices);
validStringIndicesQ @ StyleBox[ boxes_, ___] :=
  validStringIndicesQ @ boxes;
validStringIndicesQ @ other_ = False;

validStringIndexPairQ @ {_?whiteSpaceQ, _?validStringIndexQ} = True;
validStringIndexPairQ @ {_?validStringIndexQ, _?whiteSpaceQ} = True;
validStringIndexPairQ @ _ = False;

validStringIndexQ[";" | "," | _?silentParsableQ] = True;
validStringIndexQ @ other_ = False;

makeIndices @ StyleBox[ boxes_, ___] := makeIndices @ boxes;
makeIndices @ GridBox[ indices_List, ___] :=
  RowBox @ {"{", RowBox @ padList[
      parseStringIndices /@ Transpose @ indices, ","], "}"};
makeIndices @ other_ = ErrorBox @ other;

parseStringIndices @ {_?whiteSpaceQ,
    SuperscriptBox[index_?validStringIndexQ, "*"]} = index;
parseStringIndices @ {SuperscriptBox[index_?validStringIndexQ, "*"],
    _?whiteSpaceQ} = index;

parseStringIndices @
    {_?whiteSpaceQ, SuperscriptBox[index_?validStringIndexQ,
        StyleBox["*", "TensorStarStyle"]]} = index;
parseStringIndices @ {SuperscriptBox[index_?validStringIndexQ,
        StyleBox["*", "TensorStarStyle"]], _?whiteSpaceQ} = index;

parseStringIndices @ {_?whiteSpaceQ, index_?validStringIndexQ} :=
  RowBox @{"Low", "[", parseSpecial@ index, "]"};
parseStringIndices @ {index_?validStringIndexQ, _?whiteSpaceQ} :=
```

```
   RowBox @ {"High", "[", parseSpecial@index, "]"};
parseStringIndices @ other_ =
   RowBox @ {"tensorParseError", "[", other, "]"};

parseSpecial @ ";" = "\";\"";
parseSpecial @ "," = "\",\"";
parseSpecial @ other_ := First @ StripBoxes @ other

padList[list_List, padElement_] :=
   Drop[#, -1] & @ Flatten @ Thread @ {list, padElement}

whiteSpaceQ @ string_String :=
     DeleteCases[Characters @ string,
      ("\t" | "\n" | " " |
                "" | "" | "" | "" | "" | "" | "" |
               "" | "" | "\n" | "" | " " | "\:" | "␣" | "␣" | "" | "")] ≡
       {};
whiteSpaceQ @ other___ = False;

parsableQ @ boxexpr_ :=
   Head @ ToExpression[boxexpr, StandardForm, HoldComplete] ≡
      HoldComplete

SetAttributes[silentEvaluate, HoldAll];
silentEvaluate @ expr_ :=
   Block[{Message}, SetAttributes[Message, HoldFirst]; expr];
silentParsableQ @ boxexpr___ := silentEvaluate @ parsableQ @ boxexpr;

MakeExpression[TagBox[any_, Tensor, opts___], StandardForm] = $Failed;
```

D.2.3 Configurations, Permutations and Transpositions

data structure	*notation*	*usage*
`signedConfiguration[` *sign* `, {` *indices* `}]`	$\{indices\}_{\pm}$	Signed configurations represent a configuration of indices of a tensor or tensor product together with an overall sign
`SignedPermutation[` *sign* `, {` *permutation* `}]`	$\{permutation\}_{\pm}$	Signed permutations represent a permutation together with an overall sign
`SignedTransposition[` *sign*, {*i*, *j*}`]`	$(i\leftrightarrow j)_{\pm}$	Signed transpositions represent a transposition together with an overall sign. They are mathematically a subclass of signed permutations but due to our algorithm we define a special class for such objects.

```
Notation[{elms__}_+ ⟺ signedConfiguration[1, {elms__}]]
Notation[{elms__}_- ⟺ signedConfiguration[-1, {elms__}]]

Notation[configuration_+ ⟺ signedConfiguration[1, configuration_]]
Notation[configuration_- ⟺ signedConfiguration[-1, configuration_]]
Notation[configuration_sign_ ⟺ signedConfiguration[sign_, configuration_]]
```

```
Notation[{elms__}+ ⟺ SignedPermutation[1, {elms__}]]
Notation[{elms__}- ⟺ SignedPermutation[-1, {elms__}]]

Notation[(i_ ↔ j_)+ ⟺ SignedTransposition[1, {i_, j_}]]
Notation[(i_ ↔ j_)- ⟺ SignedTransposition[-1, {i_, j_}]]
Notation[(i_ ↔ j_)s_ ⟺ SignedTransposition[s_, {i_, j_}]]

Notation[⟨perms__⟩Σ=0 ⟺ LinearSymmetry[perms__]]
```

D.2.4 Standard Labeling Notation

data structure	*notation*	*usage*
`High[`*index*`, i]`	$index_i^{\uparrow}$	represents a free *contravariant* or *higher* index at position i
`Low[`*index*`, i]`	$index_i^{\downarrow}$	represents a free *covariant* or *lower* index at position i
`s[i, j]`	$s_{i\,j}$	represents a dummy index at position i summed with position j
`s[i, j, class]`	$s_{i\,j}^{class}$	represents a dummy index at position i summed with position j of class *class*
`s[i, j, High]`	$s_{i\,j}^{\uparrow}$	represents a dummy index at position i summed with position j and a fixed elevation of high
`s[i, j, Low]`	$s_{i\,j}^{\downarrow}$	represents a dummy index at position i summed with position j and a fixed elevation of low
`s[i, j, class, High]`	$s_{i\,j}^{class,\uparrow}$	represents a dummy index at position i summed with position j of class *class* and a fixed elevation of high
`s[i, j, class, Low]`	$s_{i\,j}^{class,\downarrow}$	represents a dummy index at position i summed with position j of class *class* and a fixed elevation of low

The data structures, notations, and meanings for summed indices.

```
Notation[s_i_,j_ ⟺ s[i_, j_]]
Notation[i_n_^↑ ⟺ High[i_, n_]];
Notation[i_n_^↓ ⟺ Low[i_, n_]];
Notation[s_i_,j_^↑ ⟺ s[i_, j_, High]]
Notation[s_i_,j_^↓ ⟺ s[i_, j_, Low]]
Notation[s_i_,j_^class_ ⟺ s[i_, j_, class_]]
Notation[s_i_,j_^class_,↑ ⟺ s[i_, j_, class_, High]]
Notation[s_i_,j_^class_,↓ ⟺ s[i_, j_, class_, Low]]
Notation[s_i_,j_^class_,elv_ ⟺ s[i_, j_, class_, elv_]]
```

D.3 Set Up Index Conventions

D.3.1 Indices and Index Classes

DeclareIndexClass [*class*, { i_1, i_2, ..., i_n }]	declare that the indices $i_1 \ldots i_n$ are the base indices for the index class *class*
ClassOfIndex [*index*]	return the class which *index* belongs to
ValidIndexClasses	return all the index classes in use
ValidIndexClassQ [*class*]	determine whether *class* is a valid index class
IndexQ [*index*]	determine whether *index* is an index
BaseIndices [*class*]	return all the base or root indices which belong to the index class *class*

The functions declaring and querying indices.

Set up IndexClasses.

```
ValidIndexClasses = {};
ValidIndexClassQ @ other_ = False;

ClassOfIndex @ index_Symbol+ := ClassOfIndex @ index;
ClassOfIndex @ index_Symbol- := ClassOfIndex @ index;
ClassOfIndex @ index_Symbol :=
   ClassOfIndex @ index = classOfIndexAux @ index;
```

This finds the index class of symbol for which no class has yet been specified or derived.

```
classOfIndexAux @ index_Symbol :=
   With[{ baseSymbol' = ToExpression @ baseString @ ToString @ index},
        ClassOfIndex @ baseSymbol' /; index ≠ baseSymbol']
classOfIndexAux @ index_ := IndeterminateIndexClass

ClassOfIndex @ index_ := IndeterminateIndexClass
```

This takes the first non-numeric part of a string. It is used for getting the "base string"; that is, the baseString[α4] $\longrightarrow \alpha$, baseString[ijk6u] $\longrightarrow$ ijk, etc.

```
baseString @ string_String :=
   With[
      {num = Min @ Flatten @ StringPosition[string, {"$", "1", "2", "3",
                                "4", "5", "6", "7", "8", "9", "0"}] - 1},
      If[NumberQ @ num, StringTake[string, num], string, string]]
```

We can now discern whether a symbol is an index or not.

```
IndexQ @ index_Symbol :=
   IndexQ @ index = ClassOfIndex @ index ≠ IndeterminateIndexClass
```

D.3.2 DeclareIndexClass

This piece of code first removes any indices that are already defined to be in the given class *class*. It then adds *class* to the list of valid classes. It then sets the class of each index to *class* and protects the symbol so it cannot be changed.

```
DeclareIndexClass [ class_Symbol , indices : {__Symbol}] := (
      DownValues @ ClassOfIndex = DeleteCases[
            DownValues @ ClassOfIndex, rule_ /; MemberQ[rule, class]];
      ValidIndexClasses = ValidIndexClasses ⋃ {class};
      ValidIndexClassQ @ class = True;
      BaseIndices @ class = indices;
      silentEvaluate [ Unprotect /@ indices];
      ClearAll /@ indices;
      Protect /@ indices;
      (ClassOfIndex @ elem = class) &elem /@ indices;
   )
```

Error handling for `DeclareIndexClass`.

The 1st argument of `DeclareIndexClass` must be a `Symbol`.

```
error : DeclareIndexClass[notSymb_? (headIsNot[Symbol]), ___] :=
   $Failed /;  Message[DeclareIndexClass::sym, notSymb, 1];
```

The 2nd argument of `DeclareIndexClass` must be a `List`.

```
error : DeclareIndexClass[_, notList_? (headIsNot[List]), ___] :=
   $Failed /;
      Message[DeclareIndexClass::list, HoldForm @ error, 2];
```

`DeclareIndexClass` expects 2 arguments.

```
error : DeclareIndexClass[___] :=
   With[{num = heldLength @ error}, $Failed /;
   Which[num == 1 , Message[DeclareIndexClass::argr,
                HoldForm @ DeclareIndexClass, 2],
         num ≠ 2 , Message[DeclareIndexClass::argrx,
                HoldForm @ DeclareIndexClass, num, 2],
         True,        False]];
```

D.3.3 Default Indices and Index Classes

```
End[]
```

```
"Tensors`Private`"
```

The index variables which are allowed to appear in a tensor will vary among users and also vary among disciplines. The following gives a "base" list of indices we will allow for space-time indices and also for general indices.

```
Tensors`Private`$oldContextSearchPath = $ContextPath;
$ContextPath = Tensors`Private`$contextSearchPath;

Tensors`Private`allStringIndicesUsed =
   {"a", "b", "c", "d", "i", "j", "k", "l", "m", "n", "o", "p",
      "q", "t", "α", "β", "γ", "δ", "ϵ", "ζ", "η", "λ", "μ", "ν",
      "ξ", "σ", "τ", "χ", "ω", "x", "y", "z", "r", "ρ", "θ", "ϕ"};

Tensors`Private`badIndices = Tensors`Private`silentEvaluate @ Select[
         Tensors`Private`allStringIndicesUsed,
         Head @ Context @ # ≠ Context ∧ Context @ # ≠ "Tensors`" &];

DeclareIndexClass::"overridingContexts" =
   "The symbols `1` appear outside the Tensors context. The Tensors
         package needs these symbols for indices or coordinates
         and has thus removed them from their current contexts.";

If[Tensors`Private`badIndices ≠ {},
   Message[DeclareIndexClass::"overridingContexts", StringJoin @
         Tensors`Private`padList[Tensors`Private`badIndices, ","]];
   Tensors`Private`silentEvaluate[
      ClearAll /@ Tensors`Private`badIndices;
      Remove /@ Tensors`Private`badIndices;]];

$ContextPath = Tensors`Private`$oldContextSearchPath;

DeclareIndexClass [ SpaceTimeIndices ,
   {α, β, γ, δ, ϵ, ζ, η, λ, μ, ν, ξ, σ, τ, χ, ω}]

DeclareIndexClass [ GeneralIndices , {a, b, c, d, i, j, k, l, m, n, o, p, q}

Begin @ "`Private`";

ClassOfIndex[α]

SpaceTimeIndices

BaseIndices @ SpaceTimeIndices

{α, β, γ, δ, ϵ, ζ, η, Tensors`λ, μ, ν, ξ, σ, τ, χ, ω}
```

D.3.4 Coordinates

`DeclareCoordinates[` $\{c_1, c_2, \ldots, c_n\}$`]`	declares that the c_i are to be treated as valid coordinates. Note: a coordinate cannot be an index
`CoordinateQ[`*coord*`]`	determines whether *coord* is a coordinate
`DeclareCoordinateClass[` *class*, $\{c_1, c_2, \ldots, c_m\}$`]`	declare that the coordinates of each index in the index class *class* range over the coordinates $c_1, \ldots, c_m$

The functions declaring and querying coordinates.

```
DeclareCoordinates @ coordinates_List :=
 ((
     CoordinateQ @ coordinate = True;
     IndexQ @ coordinate = False;
     If[Head @ coordinate ≡ Symbol, Protect @coordinate];) &coordinate /@
   coordinates;);

CoordinateQ @ other_ = False;

DeclareCoordinateClass[class_Symbol, {coordinates__?CoordinateQ}] :=
   (CoordinatesOfIndexClass @ class = {coordinates})
```

Error handling for `DeclareCoordinateClass`.

The 1st argument of `DeclareCoordinateClass` must be a `Symbol`.

```
error : DeclareCoordinateClass[notSymb_?(headIsNot[Symbol]), ___] :=
   $Failed /; Message[DeclareCoordinateClass::sym, notSymb, 1];
```

The 2nd argument of `DeclareCoordinateClass` must be a `List`.

```
error : DeclareCoordinateClass[_, notList_?(headIsNot[List]), ___] :=
   $Failed /;
      Message[DeclareCoordinateClass::list, error$_{hf}$, 2];

error : DeclareCoordinateClass[_, {___,
            notCoord_ /; ! CoordinateQ[notCoord], ___}, ___] := $Failed /;
      Message[DeclareCoordinateClass::nonCoordinate, notCoord$_{hf}$, error$_{hf}$];

DeclareCoordinateClass::nonCoordinate =
   "`1` is not declared as a coordinate so cannot be used in `2`";
```

`DeclareCoordinateClass` expects 2 arguments.

```
error : DeclareCoordinateClass[___] :=
   With[{num = heldLength @ error}, $Failed /;
   Which[num == 1, Message[DeclareCoordinateClass::argr,
                DeclareCoordinateClass$_{hf}$, 2],
         num ≠ 2, Message[DeclareCoordinateClass::argrx,
                DeclareCoordinateClass$_{hf}$, num, 2],
         True,        False]];
```

Error handling for `DeclareCoordinates`.

The 1st argument of `DeclareCoordinates` must be a `List`.

```
error : DeclareCoordinates[notList_?(headIsNot[List]), ___] :=
   $Failed /;
      Message[DeclareCoordinates::list, error_hf, 1];
```

`DeclareCoordinates` expects only one argument.

```
error : DeclareCoordinates[___] :=
   With[{num = heldLength @ error}, $Failed /;
      num ≠ 1 ∧ Message[DeclareCoordinates::argx,
                HoldForm @ DeclareCoordinates, num, 1]]
```

D.3.5 Default Coordinates

Set the starting coordinates the system uses to the following list.

```
End[];

DeclareCoordinates[{0, 1, 2, 3, t, x, y, z, r, ρ, θ, ϕ}]
```

The following gives a "base" list of coordinates we will allow for space-time indices and also for general indices.

```
DeclareCoordinateClass[SpaceTimeIndices, {0, 1, 2, 3}];

DeclareCoordinateClass[GeneralIndices, {1, 2, 3}];

Begin @ "`Private`";
```

D.3.6 Operations

`DeclareTensorialMultiplicative[Head[`*head*`]]`	declares that the head *head* is to be treated as a multiplicative head to be canonicalized over
`DeclareTensorialMultiplicative[Head[`*head*`], Commutivity → True | False]`	declares that the head *head* is to be treated as a multiplicative head to be canonicalized over, and treated as commutative according to the value of the option

The function for declaring that a head should be treated as multiplicative by the *Tensors* package

```
tensorialMultiplicativeHeadQ @ other_ = False;
headIsCommutativeQ @ other_ = False;
```

```
DeclareTensorialMultiplicativeHead[ head_, options___?OptionQ] :=
   Module[{commutative = Commutative /. {options} /.
               Options[DeclareTensorialMultiplicativeHead]},
      tensorialMultiplicativeHeadQ @ head = True;
      headIsCommutativeQ @ head = commutative;]

Options[DeclareTensorialMultiplicativeHead] = {Commutative → False};
```

Error Checking for `DeclareTensorialMultiplicativeHead`

The 2nd argument of `DeclareTensorialMultiplicativeHead` and beyond must be options

```
error : DeclareTensorialMultiplicativeHead[_, ___,
          notOption_?(isNot[_?OptionQ]), ___] := $Failed /;
        Message[DeclareTensorialMultiplicativeHead::nonopt,
          notOption_hf, 1, error_hf];
```

`DeclareTensorialMultiplicativeHead` expects one or more arguments

```
error : DeclareTensorialMultiplicativeHead[] := $Failed /;
        Message[DeclareTensorialMultiplicativeHead::argm,
          DeclareTensorialMultiplicativeHead_hf, 0, 1];
```

D.3.7 Default Operations

```
End[];
```

This just declares that `Times` is a commutative multiplicative head to be canonicalized over.

```
DeclareTensorialMultiplicativeHead[Times, Commutative → True];

Begin @ "`Private`";
```

D.4 The Primitive Generators

Throughout the implementation of the canonicalization algorithms, by a *primitive* tensor we mean a single indivisible factor in a tensor product. For instance, the tensor $\mathbf{R}^{\lambda\beta\gamma\tau}\,\mathbf{S}^{\mu}{}_{\nu\xi}\,\mathbf{T}^{\nu}{}_{\mu\alpha}$ is made up of the primitive tensors R, S and T.

D.4.1 Declare the Symmetries for Primitive Tensors

```
primitiveTranspositionalGenerators @ other___ = {};
closureOfPrimitiveTranspositionalGenerators @ other___ = {};
primitiveComplexGenerators @ other___ = {};
primitiveLinearGenerators @ other___ = {};
indexOptions @ other___ = {};

DeclareSymmetries[name_, length_Integer, symmetries_] :=
   DeclareSymmetries[name, {length, _, _}, symmetries];

DeclareSymmetries[name_, valence_List, symmetries_] :=
   Block[{
         S𝒯 = Cases[symmetries, _SignedTransposition],
         S𝒳 = Cases[symmetries, _SignedPermutation],
         Sℒ = Cases[symmetries, _LinearSymmetry]},

      If[S𝒯 ≠ {},
         primitiveTranspositionalGenerators[name, valence] = S𝒯];
      If[S𝒯 ≠ {}, closureOfPrimitiveTranspositionalGenerators[
               name, valence] = closeUpTranspositions @ S𝒯];
      If[S𝒳 ≠ {}, primitiveComplexGenerators[name, valence] = S𝒳];
      If[Sℒ ≠ {}, primitiveLinearGenerators[name, valence] = Sℒ];]
```

Error Handling for `DeclareSymmetries`.

The 2nd argument of `DeclareSymmetries` must be an `Integer` or a `List`.

```
error : DeclareSymmetries[_, notList_?(headIsNot[List | Integer]), ___] :=
   $Failed /;
      Message[DeclareSymmetries::listOrInteger, HoldForm@error, 2];

DeclareSymmetries::listOrInteger =
   "Integer or List expected at position `2` in `1`.";
```

The 3rd argument of `DeclareSymmetries` must be a `List`.

```
error : DeclareSymmetries[_, _, notList_?(headIsNot[List]), ___] :=
   $Failed /;
      Message[DeclareSymmetries::list, HoldForm@error, 3];
```

`DeclareSymmetries` expects 3 arguments.

```
error : DeclareSymmetries[___] :=
   With[{num = heldLength @ error}, $Failed /;
      Which[num == 1, Message[DeclareSymmetries::argr,
               HoldForm @ DeclareSymmetries, 3],
         num ≠ 3, Message[DeclareSymmetries::argrx,
               HoldForm @ DeclareSymmetries, num, 3],
         True,        False]];
```

D.4.2 Default Symmetries for some Primitive Tensors

```
DeclareSymmetries[R, 2, {(1 ↔ 2)+}]
DeclareSymmetries[R, 4, {(1 ↔ 2)-, (3 ↔ 4)-, {3, 4, 1, 2}+}]
DeclareSymmetries[A, 2, {(1 ↔ 2)-}]
DeclareSymmetries[A, 3, {(1 ↔ 2)-, (2 ↔ 3)-}]
DeclareSymmetries[A, 4, {(1 ↔ 2)-, (2 ↔ 3)-, (3 ↔ 4)-}]
DeclareSymmetries[S, 2, {(1 ↔ 2)+}]
DeclareSymmetries[S, 3, {(1 ↔ 2)+, (2 ↔ 3)+}]
DeclareSymmetries[S, 4, {(1 ↔ 2)+, (2 ↔ 3)+, (3 ↔ 4)+}]

transpositionalGenerators[S, {n_Integer, _, _}] :=
   symmetricGenerators @ n

DeclareSymmetries[R, {4, _, _}, {(1 ↔ 2)-, (3 ↔ 4)-, {3, 4, 1, 2}+,
      ⟨{1, 2, 3, 4}+, {1, 3, 4, 2}+, {1, 4, 2, 3}+⟩Σ==0}]

DeclareSymmetries[R, {4, cd_ /; cd ≥ 1, _},
   {(1 ↔ 2)-, (3 ↔ 4)-, {3, 4, 1, 2}+,
      ⟨{1, 2, 3, 4}+, {1, 3, 4, 2}+, {1, 4, 2, 3}+⟩Σ==0,
      ⟨{1, 2, 3, 4, 5}+, {1, 2, 4, 5, 3}+, {1, 2, 5, 3, 4}+⟩Σ==0}]
```

D.4.3 Characterization of Primitive Tensors

In our algorithm we need to be able to distinguish between tensors like $\mathbf{R}^{a\,b}{}_{;\,c\,d}$ and tensors like $\mathbf{R}^{a\,b}{}_{c\,d}$. One is the Ricci tensor with two covariant derivatives and one is the Riemannian tensor. Since both tensors have the same name and the same number of indices, that is R and 4, we need to include the nature or character of these indices. Therefore, the way we will distinguish between such objects internally in our algorithm is changing the name of each tensor to a fuller description or characterization of the tensor. The following function determines the kind or style or characterization of the tensor. (There are only a limited number of words to describe the concept of kind, so here I have used the word 'characterization'.) The characterization includes the number of normal indices together with the number of derivative indices. We also include the type of each index, that is a space-time index, a general index, a spin index, or maybe some fiber bundle index, etc. The characterization is only used internally in our algorithm.

```
characterizeTensor @
      Tensor[name_, indices : {n___, ";"¯, cd__, ","¯, pd__}] :=
   With[{T = tensorId[tensorCount]},
      T[name] = name; T[indexValence] = {{n}ten, {cd}ten, {pd}ten};
      T[indices] = {n, cd, pd};
      T[order] = createOrder[name, T[indexValence]]; T]
```

```
characterizeTensor @ Tensor[name_, indices : {n___, ",", pd__}] :=
   With[{T = tensorId[tensorCount]},
      T[name] = name; T[indexValence] = {{n}_len, 0, {pd}_len};
      T[indices] = {n, pd};
      T[order] = createOrder[name, T[indexValence]]; T]

characterizeTensor @ Tensor[name_, indices : {n___, ";", cd__}] :=
   With[{T = tensorId[tensorCount]},
      T[name] = name; T[indexValence] = {{n}_len, {cd}_len, 0};
      T[indices] = {n, cd};
      T[order] = createOrder[name, T[indexValence]]; T]

characterizeTensor @ Tensor[name_, indices : {n___}] :=
   With[{T = tensorId[tensorCount]},
      T[name] = name; T[indexValence] = {{n}_len, 0, 0};
      T[indices] = {n};
      T[order] = createOrder[name, T[indexValence]]; T]

createOrder[name_, valence_] := {name, Plus @@ valence} +_{} Reverse @ valence;

calculateIndexClassOfTensorIndices @ indices_ :=
   First @ Union @ Cases[ClassOfIndex /@ First /@ Flatten @ indices,
         Alternatives @@ ValidIndexClasses]
```

Using `characterizeTensor` we characterize a tensor by the name, then number of normal indices, the number of covariant derivatives, and the number of partial derivatives. Once we have characterized the tensors, we can then sort the tensors according to this characterization.

D.4.4 Closure of a Set of Transpositions

Given a set of transpositions, say $\mathcal{A}$, this calculates the set of transpositions in $\langle\mathcal{A}\rangle$, that is, the set of all transpositions which can be formed by products of the original transpositions.

```
closeUpTransposition @ generators_List :=
   generators //. {
            transpositions : {___, (i_ ↔ j_)_s_, ___, (j_ ↔ k_)_s_, ___} :>
               Sort[transpositions +_{} {(i ↔ k)_s}] /;
                  i < k ∧ ((i ↔ k)_s ∉? transpositions),

            transpositions : {___, (i_ ↔ j_)_s_, ___, (i_ ↔ k_)_s_, ___} :>
               Sort[transpositions +_{} {(j ↔ k)_s}] /;
                  j < k ∧ ((j ↔ k)_s ∉? transpositions),

            transpositions : {___, (j_ ↔ i_)_s_, ___, (k_ ↔ i_)_s_, ___} :>
               Sort[transpositions +_{} {(j ↔ k)_s}] /;
                  j < k ∧ ((j ↔ k)_s ∉? transpositions)
         } //.
      {transpositions : {___, (i_ ↔ j_)_s1_, ___, (j_ ↔ k_)_s2_, ___} :>
            (Message[DeclareSymmetries::IdenticallyZero,
                  Min[i, k], Max[i, k]]; $Failed) /; s1 == -s2,
```

```
    transpositions : {___, (i_ ↔ j_)_s1_, ___, (i_ ↔ k_)_s2_, ___} :>
        (Message[DeclareSymmetries::IdenticallyZero,
                 Min[j, k], Max[j, k]]; $Failed) /; s1 == -s2,
    transpositions : {___, (j_ ↔ i_)_s1_, ___, (k_ ↔ i_)_s2_, ___} :>
        (Message[DeclareSymmetries::IdenticallyZero,
                 Min[j, k], Max[j, k]]; $Failed) /; s1 == -s2};

DeclareSymmetries::IdenticallyZero =
   "The given tensor symmetries lead to a tensor that
         is identically zero since the transposition
         on elements `1` and `2` occurs as both a
         symmetric and anti-symmetric generator.";
```

D.4.5 Symmetric Generators and Closure of the Symmetric Generators

We need a simple function that returns the set of all possible adjacent symmetric transpositions on n indices. This is used when generating the transpositional symmetries of a tensor with multiple partial derivatives.

```
symmetricGenerators[0] = {};
symmetricGenerators[1] = {};
symmetricGenerators[n_Integer?Positive] :=
   Table[(i ↔ i + 1)_+, {i, n - 1}];
```

We also need a simple function that returns the closure of the set of adjacent symmetric transpositions on n indices. (In general the resulting set will contain non-adjacent transpositions.) This is used in checking when a tensor that is symmetrically generated is identically zero.

```
closureOfSymmetricGenerators[0] = {};
closureOfSymmetricGenerators[1] = {};
closureOfSymmetricGenerators[n_Integer?Positive] :=
   Flatten @ Table[(i ↔ j)_+, {i, 1, n - 1}, {j, i + 1, n}]
```

D.5 Constructing Generators from Primitive Generators

D.5.1 Shift Permutations

The complex and transpositional symmetries for a primitive tensor with m indices are given in terms of permutations on {1, ..., m}. When such a primitive tensor factor appears in a product tensor having n indices, then the symmetries corresponding to the primitive tensor factor need to be expressed in terms of permutations on {1, ..., n}. If the primitive tensor starts at position k in the product tensor, then the original permutation symmetries on {1, ..., m} need to be offset by k, that is, shifted so that they start with k, and then filled out in front and/or in back, to yield a permutation on {1, ..., n} that at the most moves positions only within {k, ..., $k+m$}.

```
shiftPermutation[signedPermutations_List, n_, len_] :=
   shiftPermutation[ρ, n, len] &ρ /@ signedPermutations

shiftPermutation[SignedPermutation[sign_, perm_],
      n_Integer?Positive, length_Integer?Positive] :=
   SignedPermutation[sign, Range[1, n - 1] +{}
         (perm + n - 1) +{} Range[n + perm_len, length]]

shiftPermutation[(i_ ↔ j_)sign_, n_Integer?Positive,
      len_Integer?Positive] := (i + n - 1 ↔ j + n - 1)sign

shiftPermutation[{3, 4, 1, 2}_, 5, 12]

{1, 2, 3, 4, 7, 8, 5, 6, 9, 10, 11, 12}_
```

D.5.2 Calculate Generators

We also need to be able to calculate the symmetries the tensor obeys.

```
calculateGenerators @ tensorIdentifiers_List :=
   Flatten /@ {
         calculateTranspositionalGenerators /@ tensorIdentifiers,
         calculateComplexGenerators /@ tensorIdentifiers,
         Flatten[calculateDegeneracyGenerators /@
               (T[offset] &T /@ degn &degn) /@
                  Split[tensorIdentifiers, (#1[order] ≡ #2[order]) &]],
         Flatten[calculateClosureOfTranspositionalGenerators /@
               tensorIdentifiers]}
```

```
calculateAllGenerators @ tensorIdentifiers_List :=
   Flatten /@ {
         calculateTranspositionalGenerators /@ tensorIdentifiers,
         calculateComplexGenerators /@ tensorIdentifiers,
         Flatten[calculateDegeneracyGenerators /@
               (T[offset] &T /@ degn &degn) /@
                  Split[tensorIdentifiers, (#1[order] ≡ #2[order]) &]],
         calculateLinearGenerators /@ tensorIdentifiers,
         Flatten[calculateClosureOfTranspositionalGenerators /@
               tensorIdentifiers]}
```

D.5.3 Calculate Transpositional Generators

Calculate the transpositional generators of a tensor product factor from the corresponding primitive transpositional generators. The new set of generators may contain more than just the shifted primitive generators if the tensor product involves covariant or partial derivatives of the primitive tensor.

```
calculateTranspositionalGenerators @ T_tensorId :=
   calculateTranspositionalGeneratorsAux[T, T[indexValence]]

calculateTranspositionalGeneratorsAux[T_, {n_, 0, 0}] :=
   shiftPermutation[primitiveTranspositionalGenerators[
         T[name], {n, 0, 0}], T[offset], configurationLength]

calculateTranspositionalGeneratorsAux[T_, {n_, cd_, pd_}] :=
   shiftPermutation[primitiveTranspositionalGenerators[
            T[name], {n, cd, pd}], T[offset], configurationLength] +{}
      shiftPermutation[symmetricGenerators[pd],
         T[offset] + n + cd, configurationLength]

calculateTranspositionalGeneratorsAux[T_, {0, cd_, pd_}] :=
   shiftPermutation[symmetricGenerators[cd],
         T[offset], configurationLength] +{} shiftPermutation[
         symmetricGenerators[pd], T[offset] + cd, configurationLength]
```

D.5.4 Calculate Complex Generators

Calculate the set of complex generators of a tensor product factor from the corresponding set of primitive complex generators.

```
calculateComplexGenerators@ T_tensorId :=
   shiftPermutation[primitiveComplexGenerators[
         T[name], T[indexValence]], T[offset], configurationLength]
```

D.5.5 Calculate the Closure of the Set of Transpositonal Generators

This function behaves like that of `calculateTranspositionalGenerators`, but it works on the closure of $\mathcal{T}$ rather than $\mathcal{T}$, where $\mathcal{T}$ is the set of primitive transpositional generators of the given primitive tensor.

```
calculateClosureOfTranspositionalGenerators @ T_tensorId :=
   calculateClosureOfTranspositionalGeneratorsAux[T, T[indexValence]]

calculateClosureOfTranspositionalGeneratorsAux [T_, {n_, 0, 0}] :=
   shiftPermutation[closureOfPrimitiveTranspositionalGenerators[
         T[name], {n, 0, 0}], T[offset], configurationLength]

calculateClosureOfTranspositionalGeneratorsAux [T_, {n_, cd_, pd_}] :=
   shiftPermutation[closureOfPrimitiveTranspositionalGenerators[
            T[name], {n, cd, pd}], T[offset], configurationLength] +{}
      shiftPermutation[closureOfSymmetricGenerators[pd],
         T[offset] + n + cd, configurationLength]

calculateClosureOfTranspositionalGeneratorsAux [T_, {0, cd_, pd_}] :=
   shiftPermutation[closureOfSymmetricGenerators[cd],
         T[offset], configurationLength] +{}
      shiftPermutation[closureOfSymmetricGenerators[pd],
         T[offset] + cd, configurationLength]
```

D.5.6 Calculate Degeneracy Generators

From the characterization of the factors of a tensor product, calculate the set of degeneracy generators for the overall tensor product.

```
calculateDegeneracyGenerators @ {_} = {};
```

If the head of the tensor product is not commutative, then there are no degeneracy generators.

```
calculateDegeneracyGenerators @ _ :=
   {} /; ¬ headIsCommutativeQ @ productHead
```

Finally, if there is more than one degenerate tensor and the overall product head is commutative, then calculate the degeneracy generators.

```
calculateDegeneracyGenerators[offsets_List] :=
   Inner[blockSwap[i, j] &i,j, Drop[offsets, -1], Drop[offsets, 1], List]
blockSwap[i_, j_] := SignedPermutation[1,
      Range[1, i - 1] +{} Range[j, 2 j - i - 1] +{}
         Range[i, j - 1] +{} Range[2 j - i, configurationLength]]
```

D.5.7 Calculate Linear Generators

```
calculateLinearGenerators @ T_tensorId :=
   Map[shiftPermutation[perm, T[offset], configurationLength] &perm,
      primitiveLinearGenerators[T[name], T[indexValence]], {2}]
```

D.6 Fundamental Operations

D.6.1 Permuting, Relabeling, and Group Actions

```
Notation[sp_ *π sc_ ⟺ permuteConfiguration[sp_, sc_]]
```

The following implementation, instead of using `Permute`, uses the exact same code as Permute defined in the package `DiscreteMath`Combinatorica`` but bypasses some error checking. That is, `Permute[l, p]` ≡ $l[\![p]\!]$.

```
SignedPermutation[signp_, p_] *π config_signc_ :=
   config[[inversePermutation @ p ]]signp signc ;

inversePermutation @ p_ := inversePermutation @ p =
      (Transpose @ Sort @ Transpose @ {p, Range @ pten}) [[2]] ;

generateRelabelingRules @ σ_ :=
    Dispatch @ DeleteCases[Table[i → σ[[i]], {i, σten}], (γ_ → γ_)hp]

relabelingRules @ p_ :=
   relabelingRules @ p = generateRelabelingRules @ p

relabelConfiguration[sp_SignedPermutation, config_sign_] :=
   Replace[config, relabelingRules @ sp[[2]], {2}]sign

Notation[sp_ *c sc_⟺permuteAndRelabel[sp_, sc_]]

sp_SignedPermutation *c sc_signedConfiguration :=
   relabelConfiguration[sp, sp *π sc]
st_SignedTransposition *c sc_signedConfiguration :=
   transposeAndRelabel[st, sc]

S_List *c signedConfigs_List :=
   Flatten @ Outer[permuteAndRelabel, S, signedConfigs, 1]
```

```
transposeAndRelabel[ (i_ ↔ j_)signT_ , overbar(config_signC_) ] :=
    Block[{newConfig = config},
        {newConfig[[i]] , newConfig[[j]]} = config[[{j, i}]];
        overbar(Replace[newConfig, {i → j, j → i}, {2}]signT signC) ]
```

D.6.2 Ordering of Indices and Configurations

```
InfixNotation[≶c, signedConfigurationOrder]

overbar(config1_sign1_) ≶c overbar(config2_sign2_) :=
  Block[{ordering},
    ordering = Order[Sort @ Cases[config1, _High],
                     Sort @ Cases[config2, _High]];
    If[ordering == 0, ordering = Order[Reverse @ Sort @ Cases[config2, _Low],
                                       Reverse @ Sort @ Cases[config1, _Low]];
      If[ordering == 0, ordering = Order[ config1 /. s → elevation,
                                          config2 /. s → elevation];
        If[ordering == 0, ordering = Order[ config1 /. index_s :↦ index[[2]],
                                            config2 /. index_s :↦ index[[2]] ];
          If[ordering == 0, ordering = Order[
                                  config1 /. s → classOfSummedIndex,
                                  config2 /. s → classOfSummedIndex]]]]];
    ordering]
```

The class of an index is determined by `classOfSummedIndex`. Any index without a specified class is treated as a general index.

```
classOfSummedIndex[_, _] := General;
classOfSummedIndex[_, _, High | Low] = General;
classOfSummedIndex[_, _, class_] = class;
classOfSummedIndex[_, _, class_, _] = class;
```

The elevation of a summed index is the fixed elevation if given, or `High` if it is the first of a summed pair and Low if it is the second of a summed pair.

```
elevation[i_Integer, j_Integer] := If[i < j, High, Low];
elevation[_, _, _, elevation_] = elevation;
elevation[_, _, elevation : (High | Low)] = elevation;
elevation[i_Integer, j_Integer, class_] = If[i < j, High, Low];

InfixNotation[≺c, signedConfigurationLess]
InfixNotation[≼c, signedConfigurationLessEqual]
InfixNotation[≽c, signedConfigurationGreaterEqual]
InfixNotation[≻c, signedConfigurationGreater]
InfixNotation[==c, signedConfigurationEquivalent]
```

For two arguments:

```
a_ ≺c b_ := (a ≶c b) == 1
a_ ≼c b_ := (a ≶c b) ≥ 0
```

```
a_ ⩾c b_ := (a ≶c b) ≤ 0
a_ >c b_ := (a ≶c b) == -1
a_ ==c b_ := (a ≶c b) == 0
```

For three or more arguments:

```
With[{sc = signedConfiguration, sco = signedConfigurationOrder},
   f_sc <c mid__sc <c g_sc := Inner[sco, {f, mid}, {mid, g}, Min] == 1;
   f_sc ⩽c mid__sc ⩽c g_sc := Inner[sco, {f, mid}, {mid, g}, Min] ≥ 0;
   f_sc ⩾c mid__sc ⩾c g_sc := Inner[sco, {f, mid}, {mid, g}, Max] ≤ 0;
   f_sc >c mid__sc >c g_sc := Inner[sco, {f, mid}, {mid, g}, Max] == -1;
   f_sc ==c mid__sc ==c g_sc :=
      Inner[sco, {f, mid}, {mid, g}, Union @ List @ # &] == {0};]
```

In the special case when the free indices are in their canonical positions in both signed configurations, then we no longer have to perform the comparison between the free indices since they will be the same.

```
config1_sign1_ ≶c config2_sign2_ /; freeIndicesAreCanonical :=
   Block[{ordering},
      ordering = Order[config1 /. s → elevation, config2 /. s → elevation];
      If[ordering == 0, ordering =
            Order[config1 /. index_s :↦ index[[2]], config2 /. index_s :↦ index[[2]]];
         If[ordering == 0, ordering = Order[config1 /. s → classOfSummedIndex,
               config2 /. s → classOfSummedIndex]]];
      ordering]
```

D.6.3 Minimum Configurations

```
Symbolize[Minc, SymbolizeRootName → "minimumConfiguration"]

Minc @ {signedConfig_signedConfiguration} = signedConfig;
Minc @ signedConfigs_List := Module[{smallest = signedConfigs[[1]]},
      Scan[If[sc <c smallest, smallest = sc] &sc, signedConfigs]; smallest]
```

D.7 Converting Tensor Products to Configurations

D.7.1 Encoding Tensor Products into Configurations

At the beginning of the algorithm, we need to manipulate the tensors into a form for the canonicalization algorithm. This means we must sort the tensors if required, transform the list of indices to a configuration, and store all pertinent information about the tensors such as: tensor names, number of normal indices, number of partial derivatives, number of covariant derivatives, etc. This information is used to create the set of symmetries the tensor product will obey and also allows us to restore the signed configuration back to a tensor when we have obtained the canonical form.

```
Options @ encodeTensors = {sortTensors → Automatic,
      IndexClassInclusion → False, MetricLocallyFlat → False};

encodeTensors[ tensor_Tensor, options____?OptionQ] :=
   {Identity} +{} encodeTensorsAux[ {tensor}, options]
encodeTensors[ head_ @ tensors__Tensor, options____?OptionQ] :=
   {head} +{} encodeTensorsAux[head @ tensors, options];

encodeTensorsAux[ head_ @ tensors___, options____?OptionQ] :=
   Block[{sortedTensors, sortedTensorIds,
         tensorIdentifiers, tensorCount, indexLists, lengthsOfIndexLists,
         indexOffsets, labeledConfig, signedConfiguration,
         sortThem = sortTensors /. {options} /. Options[encodeTensors],
         optMetricLocallyFlat =
            MetricLocallyFlat /. {options} /. Options[encodeTensors],
         IndexClassInclusion = IndexClassInclusion /. {options} /.
               Options[encodeTensors]},

      (* Store the information about the tensors *)
      Clear @ tensorId;
      tensorCount = 0;
      tensorIdentifiers =
          (++tensorCount; characterizeTensor @ #) & /@ {tensors};

      (* Sort the tensors and calculate the offsets of each tensor *)
      If[sortThem ≡ Automatic, sortThem = headIsCommutativeQ @ head];
      sortedTensorIds = If[sortThem, Sort[tensorIdentifiers,
                (#1[order] ≤lex #2[order]) &], tensorIdentifiers];

      indexLists = T[indices] &T /@ sortedTensorIds;
```

```
    lengthsOfIndexLists = Length /@ indexLists;
    indexOffsets = Drop[FoldList[Plus, 1, lengthsOfIndexLists], -1];
    Inner[(T[offset] = offset) &T,offset, sortedTensorIds, indexOffsets, List];

    (* Label the configuration *)
    labeledConfig =
      labelTheConfiguration[Join @@ indexLists, sortedTensorIds];
    signedConfiguration = labeledConfig+;

    {sortedTensorIds, signedConfiguration}]

SetAttributes[{encodeTensors, encodeTensorsAux}, HoldFirst]
```

This creates a signed labeled configuration.

```
labelTheConfiguration[theIndices_, sortedTensorIds_] :=
  Block[{labeledConfig, fixedElevationPositions, allIndices = theIndices},

    (* Wrap any numeric coordinates.*)
    labeledConfig = wrapNumericCoordinates /@ allIndices;

    (* Add positions to the indices.*)
    labeledConfig = MapIndexed[addPosition , labeledConfig];

    (* Change summed indices to s[i,j] indices.*)
    labeledConfig = transformToSijLabels @ labeledConfig;

    (* If the metric is not locally flat we must fix the elevations at certain positions. *)
    If[ optMetricLocallyFlat == True , labeledConfig,
      (* Fix the elevation of specific indices.*)
      fixElevationOfTensorIndices /@ sortedTensorIds;
      labeledConfig]]
```

The above code used the following auxiliary procedure that just affixes the position of the index to the index.

```
addPosition[α_+, {n_}] = α_n^↑;
addPosition[α_-, {n_}] = α_n^↓;
```

When we use numeric coordinates like 0, 1, 2, 3, we must include a wrapper around them so they are not transformed under permutation and relabeling.

```
wrapNumericCoordinates @ α_Integer+ := coordinateWrapper[α]+;
wrapNumericCoordinates @ α_Integer- := coordinateWrapper[α]-;
wrapNumericCoordinates @ other_ = other;
```

In special circumstances, certain types of indices are not allowed to be raised and lowered in pairs at liberty. This is fully explained in §6.12 *Refinements for Partial Derivatives*. As a default, we assume the connection is a metric one, which implies that indices appearing inside covariant derivatives can be raised and lowered in pairs. However, in general this is not true for indices appearing inside partial derivatives, so we fix these indices. Therefore, as a general default, we assume the metric is not locally flat; however, we can override this with specific options.

```
optMetricLocallyFlat = False;
```

```
fixElevationOfTensorIndices @ T_tensorId :=
   fixElevationOfTensorIndicesAux[T[indexValence], T[offset]];
fixElevationOfTensorIndicesAux[{n_, c_, 0}, offset_] = {};
fixElevationOfTensorIndicesAux[{n_, c_, _}, offset_] :=
   Block[{start = offset, finish = offset + n + c - 1, index},
      Do[fixElevationOfIndex @ labeledConfig[[pos]], {pos, start, finish}]]
```

For any index summed with a corresponding index outside the tensor which is being partially differentiated, we need to fix the elevations of both indices. If the summed indices both occur within the tensor, we do not need to fix their elevations. See §6.12 *Refinements for Partial Derivatives* for further explanations.

```
fixElevationOfIndex @ s_[i_,j_] /; (j < start ∨ j > finish) := (
      labeledConfig[[i]] = s[i, j, Head @ allIndices[[i]]];
      labeledConfig[[j]] = s[j, i, Head @ allIndices[[j]]];);

fixElevationOfIndex @ s_[i_,j_]^?ValidIndexClassQ /; (j < start ∨ j > finish) := (
      labeledConfig[[i]] = s[i, j, class, Head @ allIndices[[i]]];
      labeledConfig[[j]] = s[j, i, class, Head @ allIndices[[j]]];);

fixElevationOfIndex @ other_ = False;
```

D.7.2 Transform to S[i,j] Labels

The following function just transforms all pairs of dummy labels into $s_{i,j}$ type labels. It is somewhat cryptically coded in an effort to achieve speed. It turns out that this function is one of the slowest parts of encoding a tensor, and usually takes about 1/3 of the overall time to canonicalize a tensor. (The overall algorithm is still very fast: ~ 0.1-0.2 of a second on a 160Mhz 604e based Macintosh.) However, the encoding process must be run on every term in an expression. So if one has, say, a hundred terms, then we have already taken in the order of 10 to 20 seconds just to encode them. Thus it is important to have this routine as fast as possible.

```
transformToSijLabels @ labeledConfig_ :=
   sortByPosition @
      Flatten[labelPair /@
            Split[sortByIndex @ labeledConfig, (#1[[1]] ≡ #2[[1]]) &]]

sortByIndex @ labeledConfig_ := #[[1]] & /@ Sort[(#[[1]])[#] & /@ labeledConfig];
sortByPosition @ labeledConfig_ :=
   #[[1]] & /@ Sort[(pos @ #)[#] & /@ labeledConfig];

pos @ s_[i_,_] = i;
pos @ _i_^↑ = i;
pos @ _i_^↓ = i;
```

If we are including index classes in our summed indices, then include the relavent index class when we label a pair of indices.

```
labelPair @ {α_i_↑, α_j_↓} :=
    {s_{i,j}^(ClassOfIndex @ α), s_{j,i}^(ClassOfIndex @ α)} /; IndexClassInclusion ∧ IndexQ @ α;
labelPair @ {α_i_↓, α_j_↑} :=
    {s_{i,j}^(ClassOfIndex @ α), s_{j,i}^(ClassOfIndex @ α)} /; IndexClassInclusion ∧ IndexQ @ α;

labelPair @ {α_i_↑, α_j_↓} := {s_{i,j}, s_{j,i}} /; IndexQ @ α;
labelPair @ {α_i_↓, α_j_↑} := {s_{i,j}, s_{j,i}} /; IndexQ @ α;
labelPair @ other_ = other;
```

Note: a simplistic version might be something like the following, which, however, fails on several counts. One, it is slower and two, it might fail if there are three indices to be summed. Currently, the canonicalization algorithm leaves three or more repeated indices alone.

```
transformToSijLabels @ labeledConfig_ :=
    labeledConfig //.
        {{r___, α_i_↑, m___, α_j_↓, l___} → {r, s_{i,j}, m, s_{j,i}, l},
            {r___, α_j_↓, m___, α_i_↑, l___} → {r, s_{i,j}, m, s_{j,i}, l}}
```

D.7.3 Encoding Examples

Here is the tensor encoded into a signed configuration.

```
encodeTensors[{R^(α β γ δ), R^ν_(ε τ γ, a), A^μ_(ν ξ β σ)}, sortTensors → False]
```

```
{List, {tensorId[1], tensorId[2], tensorId[3]},
    {{α_1^↑, s_{2,13}, s_{3,8}^↑, δ_4^↑, s_{5,11}^↑, ε_6^↓, τ_7^↓, s_{8,3}^↓, a_9^↓, μ_10^↑, s_{11,5}^↓, ξ_12^↓, s_{13,2}, σ_14^↓}_+}}
```

```
encodeTensors[{R^(α β γ δ), R^ν_(ε τ γ, a), A^μ_(ν ξ β σ)},
    sortTensors → False, MetricLocallyFlat → True]
```

```
{List, {tensorId[1], tensorId[2], tensorId[3]},
    {{α_1^↑, s_{2,13}, s_{3,8}, δ_4^↑, s_{5,11}, ε_6^↓, τ_7^↓, s_{8,3}, a_9^↓, μ_10^↑, s_{11,5}, ξ_12^↓, s_{13,2}, σ_14^↓}_+}}
```

D.8 Converting Configurations to Tensor Products

D.8.1 Reconstitute Tensors

Once we have a canonical configuration, we can reconstitute the indices as above. Then finally, using the characterized tensor names, we can form the new canonical tensor product. First, we need to be able to calculate which class an index comes from. If this is not readily apparent, then use the default class for that particular tensor; and if there is no default for that tensor, then use the general default.

```
determineIndexClasses @ T_tensorId :=
   ClassOfIndex /@ T[indices] /. indexOptions @ T[name]
```

Now we are in a position to reconsitute a labeled signed configuration back into a tensor.

```
reconstituteTensors[head_, tensorIdentifiers_List, configuration_sign_] :=
   Block[{reconstitutedIndices, indexClasses},

      (* Determine the classes of the indices in the original tensor *)
      indexClasses = Flatten[determineIndexClasses /@ tensorIdentifiers];

      (* Reconstitute s[i,j] indices to real indices *)
      reconstitutedIndices = reconstituteIndices @ configuration;

      (* Update the indices of the tensors, return the reconstructed tensor *)
      (T[indices] = Take[reconstitutedIndices,
                        {T[offset], T[offset] - 1 + Plus @@
                                 T[indexValence]}]) &T /@ tensorIdentifiers;
      sign * (head @@ reconstructTensor /@ tensorIdentifiers)]
```

This code reconstructs a tensor from the characterized parts of a tensor identifier. This is the inverse of characterization.

```
reconstructTensor @ T_tensorId :=
   Block[{n, cd, pd, indices = T[indices]},
      {n, cd, pd} = T[indexValence];
      tensorIndices = Take[indices, n] +{}
            If[cd ≠ 0, {";"}, {}] +{} Take[indices, {n + 1, n + cd}] +{}
            If[pd ≠ 0, {","}, {}] +{} Take[indices, {n + 1 + cd, n + cd + pd}];
      Tensor[T[name], tensorIndices]]
```

D.8.2 Reconstitute Indices

At the end of our canonicalization algorithm, we need to transform our canonical signed configuration using $s_{i,j}$ indices back into a signed configuration with normal index labels like α, β, i, j, k, m, etc. To do this, we *reconstitute* the indices. All the new indices generated will be of the class from which the index came. Also, some indices cannot change their covariant or contravariant nature, so if an index is fixed to be high or low, we must be aware of this. `reconstituteIndices` assumes that *indexClasses* has been dynamically set in the code that calls it.

```
reconstituteIndices @ configuration_ :=
  Block[{indexTable, reconstitutedIndices},

    (* Initialize index table *)
    indexTable @ usedIndices = Cases[configuration, α_^↑ | α_^↓ → α];

    (* Reconstitute the indices *)
    reconstitutedIndices = reconstituteIndex /@ configuration;

    (* Make a second pass for the partners of the summed indices *)
    reconstitutedIndices =
      reconstitutedIndices /. reconstituteIndex → oppositeIndex;

    (* Remove any coordinate wrapers *)
    reconstitutedIndices /. coordinateWrapper @ any_ → any]
```

Following is the supporting code for `reconstituteIndices`. We use an *indexTable* in order to store the lists of available indices.

```
reconstituteIndex @ α_^↑ = α^+;
reconstituteIndex @ α_^↓ = α^-;
reconstituteIndex @ s_{i_,j_} /; i < j := High @ getIndex @ indexClasses[[i]];
reconstituteIndex @ s^↑_{i_,j_} /; i < j := High @ getIndex @ indexClasses[[i]];
reconstituteIndex @ s^↓_{i_,j_} /; i < j := Low @ getIndex @ indexClasses[[i]];
reconstituteIndex @ s^class_{i_,j_} /; i < j := High @ getIndex @ class;
reconstituteIndex @ s^{class_,↑}_{i_,j_} /; i < j := High @ getIndex @ class;
reconstituteIndex @ s^{class_,↓}_{i_,j_} /; i < j := Low @ getIndex @ class;

oppositeIndex @ s_{j_,i_} := oppositeElevation @ reconstitutedIndices[[i]];
oppositeElevation @ α_^+ = α^-;
oppositeElevation @ α_^- = α^+;
```

D.8.3 getIndex

The set BaseIndices[*class*] gives only a finite number of acceptable indices. However, we may well need more than these. It seems reasonable that if x is a base index, then x1, x2, ... should be acceptable indices. In line with generating extra indices besides the starting ones, we next define the function AppendToName so that, given any integer, for example 8, then AppendToName[8] will itself be a function which applied to any symbol, for example α, will give the symbol $\alpha 8$.

The following code is inelegant. And though mundane, it yet involves necessary book-keeping of which indices can be used or reused, etc. It is not immediately apparent how to make this code more elegant.

```
appendToName[m_][symb_Symbol] :=
   ToExpression[ToString @ symb <> ToString @ m];

indicesInClassForCount[class_, n_] := indicesInClassForCount[class, n] =
      appendToName[n][symb] &symb /@ BaseIndices @ class
```

It should be noted that before getIndex is called, *indexTable* should not have any values in it. This is obeyed inside this package by having *indexTable* be dynamically scoped inside a block on each occasion when it is used to generate a new set of indices.

```
getIndex @ IndeterminateIndexClass := (
      Message[getIndex::IndeterminateIndex];
      getIndex @ GeneralIndices)

getIndex::IndeterminateIndex =
   "getIndex has been called with an index of
         IndeterminateIndexClass. A general index is
         being used. This has most likely occurred during
         the reconsitution of a tensor product from a
         configuration. To avoid this problem set the option
         IndexClassInclusion → True when calling Canonicalize.";

getIndex @ class_ /; indexTable @ class ≠ {} :=
   Block[{newIndex = First @ indexTable @ class},
      indexTable @ class = Drop[indexTable @ class, 1];
      newIndex]

getIndex @ class_ /; ¬ ValueQ @ indexTable @ class := (
      indexTable[class, count] = 1;
      indexTable @ class =
         BaseIndices @ class \⧸ indexTable @ usedIndices;
      getIndex @ class)

getIndex @ class_ /; indexTable @ class ≡ {} :=
   Block[{newIndex},
      indexTable @ class =
         indicesInClassForCount[class, indexTable[class, count]] \⧸
```

```
        indexTable @ usedIndices;
    ++indexTable[class, count];
    getIndex @ class]
```

D.8.4 Reconstitution Examples

```
encodeTensors[{R^{αβγτ}, R^{0}_{ετγ,a}, A^{μ}_{νξ,b}}, sortTensors → True]
```

```
{List, {tensorId[3], tensorId[1], tensorId[2]},
  {μ_1^↑, ν_2^↓, ξ_3^↓, b_4^↓, α_5^↑, β_6^↑, s_{7,12}^↑, s_{8,11}^↑, coordinateWrapper[0]_9^↑, ε_10^↓, s_{11,8}^↓, s
  }
```

```
reconstituteTensors @@ %
```

```
{A^{μ}_{νξ,b}, R^{αβγδ}, R^{0}_{εδγ,a}}
```

```
encodeTensors[{R^{αβγτ}, R^{0}_{ετγ,a}, A^{μ}_{νξ,b}},
  sortTensors → True, indexKindInclusion → True]
```

```
{List, {tensorId[3], tensorId[1], tensorId[2]},
  {μ_1^↑, ν_2^↓, ξ_3^↓, b_4^↓, α_5^↑, β_6^↑, s_{7,12}^↑, s_{8,11}^↑, coordinateWrapper[0]_9^↑, ε_10^↓, s_{11,8}^↓, s
  }
```

```
reconstituteTensors @@ %
```

```
{A^{μ}_{νξ,b}, R^{αβγδ}, R^{0}_{εδγ,a}}
```

D.9 The Basic Canonicalizing Algorithm

D.9.1 The Basic Canonicalizing Algorithm

```
BasicCanonicalize @ tensors_Plus := BasicCanonicalize /@ tensors;
BasicCanonicalize @ (num_?NumberQ tensorProduct_) :=
  num BasicCanonicalize @ tensorProduct;
BasicCanonicalize @ other_ = other;

BasicCanonicalize @ tensorProduct : (head_ @ __Tensor | _Tensor) :=
  Block[{S, productHead, tensorIdentifiers, signedConfiguration,
      configurationLength, allEquivalentConfigurations, tensorId},
```

```
        {productHead, tensorIdentifiers, signedConfiguration} =
            encodeTensors @ tensorProduct;
        configurationLength = signedConfiguration[[2]]ten;
        S = Union @ Flatten @ calculateGenerators @ tensorIdentifiers;
        allEquivalentConfigurations = generateConfigurations[signedConfiguration, S];
        If[ tensorIdenticallyZeroQ[allEquivalentConfigurations], 0,
            reconstituteTensors[productHead,
                tensorIdentifiers, Min_c @ allEquivalentConfigurations]]]

tensorIdenticallyZeroQ @ signedConfigs_List :=
    Union[sc[[2]] &_sc /@ signedConfigs]_ten < signedConfigs_ten
```

D.9.2 Generating Configurations

```
generateConfigurations[seedConfiguration_signedConfiguration, S_List] :=
    Block[{F = {}, N = {seedConfiguration}},
        While[N ≠ {},
            F = F ⋃ N;
            N = (S *_c N) \ F];
        F]
```

D.9.3 Examples of Basic Canonicalize

Omitted.

D.10 The Optimized Canonicalizing Algorithm

D.10.1 Transpositional Canonicalization Operator

```
Notation[a_ ⇆? b_ ⇔ reducingSwapQ[a_, b_]]

_s ⇆? _High = True;
_High ⇆? _s = False;
_s ⇆? _Low = False;
_Low ⇆? _s = True;
_High ⇆? _Low = False;
_Low ⇆? _High = True;
```

```
b_↑ ⇆? a_↑ := a ≺lex b;
b_↓ ⇆? a_↓ := a ≺lex b;
s_i_,m_ ⇆? s_j_,n_ = i < j < n < m ∨ n < m < i < j ∨ Max[i, m] < Min[j, n];

Notation[Θ_T_[c_] ⟺ transpositionallyCanonicalize[c_, T_]]

Θ_T_List[signedConfigs_List] := Θ_T[sc] &sc /@ signedConfigs;
Θ_T_List[configuration_signC_] :=
    Block[
      {newConfiguration = configuration, newSign = signC, partialCanonicalConfiguration = {}},
        While[partialCanonicalConfiguration ≠ newConfiguration,
            partialCanonicalConfiguration = newConfiguration;
            transposeIfMoreCanonical /@ T];
      partialCanonicalConfiguration_newSign]

transposeIfMoreCanonical @ (i_ ↔ j_)_signτ_ :=
   If[newConfiguration_[[i]] ⇆? newConfiguration_[[j]], swapThem[signτ, i, j]]

swapThem[signτ_, i_, j_] := (
      {newConfiguration_[[i]], newConfiguration_[[j]]} = newConfiguration_[[{j, i}]];
      newSign = newSign signτ;
      newConfiguration = Replace[newConfiguration, {i → j, j → i}, {2}]);

compareS = Compile[
      {{i, _Integer}, {j, _Integer}, {m, _Integer}, {n, _Integer}},
      Max[i, m] < Min[j, n] ∨ i < j < n < m ∨ n < m < i < j ];
```

Actually the following code is faster.

```
compareS = Compile[
      {{i, _Integer}, {j, _Integer}, {m, _Integer}, {n, _Integer}},
      (i < j ∧ (m < n ∧ (j < n ∧ (m < i ∨ i < m ∧ m < j) ∨ n < j ∧ (i < m ∨ m < i ∧ i < n)
                  n < m ∧ (j < n ∨ m < i)))];

s_i_,m_ ⇆? s_j_,n_ := compareS[i, j, m, n];
```

D.10.2 GenerateConfigurations$_{\mathcal{T}}$

```
Notation[generateConfigurations_T_[sc_, R_] ⟺
      generateTranspositionallyCanonicalConfigurations[sc_, R_, T_]]

generateConfigurations_T_List[
      seedConfiguration_signedConfiguration, R_List] :=
   Block[{F = {}, N = {Θ_T[seedConfiguration]}},
      While[N ≠ {},
          F = F ⋃ N;
          N = Θ_T[R *_c N] \ F;];
      F]
```

D.10.3 Canonicalization of Free Indices

```
canonicalizeFreeIndices[
      sc_signedConfiguration, S_T : _, S_X : _, S_D : _] :=
   Block[{n = sc, final = Null, t},
      While[final ≠ n,
         final = n;
         If[(t = ρ *_c n) <_c n, n = t] &_ρ /@ S_D;];
      final = Null;
      While[final ≠ n,
         final = n = ⊙_{S_T}[n];
         If[(t = ρ *_c n) <_c n, n = t] &_ρ /@ S_X];
      final]
```

D.10.4 Stabilize Permutations

Once we have put the free indices into their canonical positions, we only need to generate the stabilized subgroup of permutations which keep the free indices fixed. Since the free indices are the most important in our ordering, we know that any comparison of configurations is first determined on the free indices and then on the summed indices. This means that if the free indices of a configuration $a \in C$ are in their canonical positions with respect to the group $\mathcal{G}$, then the minimum configuration of $\mathcal{G}_{stab} *_c a$ will be the same as the minimum configuration of $\mathcal{G} *_c a$ (where $\mathcal{G}_{stab}$ is the subgroup of $\mathcal{G}$ which does not move the free indices of the configuration a).

Because of the properties of our generating set $\mathcal{S}$, we can generate $\mathcal{S}_{stab}$ simply by removing from $\mathcal{S}$ all $s \in \mathcal{S}$ such that s moves one or more of the free indices of the configuration a which has its free indices in canonical positions. We can also remove all elements that are equivalent to some other element.

```
stabilizePermutations[configuration_sign_, S_T : _, S_X : _, S_D : _] :=
   Block[{stabilizedPositions = Cases[configuration, _n_^↑ | _n_^↓ → n],
         stabilizedT, stabilizedX, stabilizedD},
      stabilizedT = removePermutationsWhichMoveIndices[
            S_T, stabilizedPositions];
      stabilizedD = removePermutationsWhichMoveIndices[
            S_D, stabilizedPositions];
      stabilizedX = removePermutationsWhichMoveIndices[
            S_X, stabilizedPositions ⋃ Flatten[
                  (Select[ρ, ρ_[[m]] < m &_m] &_ρ) /@ (σ_[[2]] &_σ) /@ stabilizedD]];
      {stabilizedT, stabilizedX, stabilizedD}]
```

```
removePermutationsWhichMoveIndices[S_List, {}] = S;
removePermutationsWhichMoveIndices[S_List, {m_, rest___}] :=
   removePermutationsWhichMoveIndices[
      DeleteCases[S, (SignedPermutation[_, p_] /; p[[m]] ≠ m) |
            (m ↔ _)_ | (_ ↔ m)_], {rest}]
```

Also, we are sometimes able to remove some excess $\mathcal{X}$ generators (complex generators), since if the $\mathcal{D}$ generators are still present after stabilization, we will have an overcomplete generating set. For instance, if $\mathcal{X} = \{\sigma, \pi\}$ where $\pi = \{3, 4, 1, 2, 5, 6, 7, 8\}$ and $\sigma = \{1, 2, 3, 4, 7, 8, 5, 6\}$ and $\rho = \{5, 6, 7, 8, 1, 2, 3, 4\}$ is part of $\mathcal{D}$, then we can remove σ from $\mathcal{X}$ since ρ and π are sufficient. That is, $\sigma = \rho \cdot \pi \cdot \rho^{-1}$, and so ρ and π generate the same set of configurations as ρ, π and σ. Of course, we always need the full set of adjacent transpositions.

To remove excess $\mathcal{X}$ generators (complex generators), we simply find all the larger indices that are moved by the degenerate name symmetries and remove all permutations in $\mathcal{X}$ that move these indices.

D.10.5 Identically Zero Tensors

```
Symbolize[i_ind]; Symbolize[j_ind]

inducedTransposition @ (i_ ↔ j_)_sign_ :=
   Block[{
         i_ind = i /. inducingRules,
         j_ind = j /. inducingRules},
      If[i_ind < j_ind, (i_ind ↔ j_ind)_sign, (j_ind ↔ i_ind)_sign]]

tensorIdenticallyZeroQ[
      minSignedConfiguration_, signedConfigs_List, T■ : _List] :=
   Block[
      {allT, minConfiguration = minSignedConfiguration[[2]], inducingRules},
      inducingRules = Flatten @ Cases[minConfiguration, s_i_,j_ → {i → j}];
      allT = T■ ⋃ (inducedTransposition /@ T■);
      (Union[sc[[2]] &sc /@ signedConfigs]_len < signedConfigs_len) ⋁
         (Union[st[[2]] &st /@ allT]_len < allT_len) ⋁
         directlyZeroQ[minConfiguration, T■]]

directlyZeroQ [ minConfiguration_, T■ : _List] :=
   Block[{reducedIndices = reduceIndex /@ minConfiguration},
      Or @@ directlyZeroTestAuxQ /@ T■]

directlyZeroTestAuxQ @ (i_ ↔ j_)_+ = False;
directlyZeroTestAuxQ @ (i_ ↔ j_)_ := reducedIndices[[i]] ≡ reducedIndices[[j]]

reduceIndex @ s_i_,j_ := s @ Min[i, j];
reduceIndex @ h_[index_, ___] = h @ index;
```

D.10.6 Optimized Canonicalize

```
OptimizedCanonicalize @ tensors_Plus := OptimizedCanonicalize /@ tensors
OptimizedCanonicalize @ (num_ tensorProduct_) :=
   num OptimizedCanonicalize @ tensorProduct /; FreeQ[num, Tensor];
OptimizedCanonicalize @ other_ = other;

OptimizedCanonicalize @ tensorProduct : (head_ @ __Tensor | _Tensor) :=
   Block[{S_T, S_X, S_D, (S_T)_., productHead, tensorIdentifiers, signedConfiguration,
         allEquivalentConfigurations, tensorId, configurationLength},
      {productHead, tensorIdentifiers, signedConfiguration} =
         encodeTensors @ tensorProduct;
      configurationLength = signedConfiguration[[2]]_len;
      {S_T, S_X, S_D, (S_T)_.} = calculateGenerators @ tensorIdentifiers;
      signedConfiguration =
         canonicalizeFreeIndices[signedConfiguration, S_T, S_X, S_D];
      {S_T, S_X, S_D} = stabilizePermutations[signedConfiguration, S_T, S_X, S_D];
      allEquivalentConfigurations =
         generateConfigurations_{S_T}[signedConfiguration, S_D ⋃ S_X];
      signedConfiguration = Min_c @ allEquivalentConfigurations;
      If[tensorIdenticallyZeroQ[
            signedConfiguration, allEquivalentConfigurations, (S_T)_.], 0,
         reconstituteTensors[productHead, tensorIdentifiers, signedConfiguration]]]
```

D.10.7 Canonicalize Brief

```
CanonicalizeBrief @ tensors_Plus := CanonicalizeBrief /@ tensors
CanonicalizeBrief @ (num_?NumberQ tensorProduct_) :=
   num CanonicalizeBrief @ tensorProduct
CanonicalizeBrief @ other_ = other

other

CanonicalizeBrief @ tensorProduct : (head_ @ __Tensor | _Tensor) :=
   Block[{S_T, S_X, S_D, (S_T)_., productHead,
         tensorIdentifiers, signedConfiguration, tensorId, configurationLength},
      {productHead, tensorIdentifiers, signedConfiguration} =
         encodeTensors @ tensorProduct;
      configurationLength = signedConfiguration[[2]]_len;
      {S_T, S_X, S_D, (S_T)_.} = calculateGenerators @ tensorIdentifiers;
      signedConfiguration =
         canonicalizeFreeIndices[signedConfiguration, S_T, S_X, S_D];
      reconstituteTensors[productHead, tensorIdentifiers, signedConfiguration]]
```

D.10.8 Optimized Canonicalization Examples

Omitted.

D.11 Refinements for Fixed Elevation Indices

D.11.1 Refined Transpositional Canonicalization Operator: Fixed Elevations

This code is predicated on §C.2 *Evidence for Steepest Descent Conjecture*. It is necessary if we use tensors containing partial derivatives, since then we cannot raise and lower indices at free will.

Clearly we should not swap $\uparrow \leftrightarrows \downarrow$ but we should swap $\downarrow \leftrightarrows \uparrow$.

$$s^{\uparrow}_{i\_,m\_} \leftrightarrows_? s^{\downarrow}_{j\_,n\_} = \texttt{False};$$
$$s^{\downarrow}_{i\_,m\_} \leftrightarrows_? s^{\uparrow}_{j\_,n\_} = \texttt{True};$$

Also, if both signs are the same, then we should clearly just order using the normal conventions.

$$s^{\uparrow}_{i\_,m\_} \leftrightarrows_? s^{\uparrow}_{j\_,n\_} = n < m;$$
$$s^{\downarrow}_{i\_,m\_} \leftrightarrows_? s^{\downarrow}_{j\_,n\_} = n < m;$$

And finally, when we mix kinds, we have the following.

$$s^{\downarrow}_{i\_,m\_} \leftrightarrows_? s_{j\_,n\_} = m > n \vee n > i;$$
$$s^{\uparrow}_{i\_,m\_} \leftrightarrows_? s_{j\_,n\_} = j < n < m;$$

$$s_{i\_,m\_} \leftrightarrows_? s^{\downarrow}_{j\_,n\_} = n < m < i;$$
$$s_{i\_,m\_} \leftrightarrows_? s^{\uparrow}_{j\_,n\_} = j > m \vee m > n;$$

To show that this code actually fulfills its design, we again have to conjecture that transpositional canonicalization can again be achieved in a downhill fashion. As before, we again have to defer to computer justification of our conjecture, and this is given in §C.2 *Evidence for Steepest Descent Conjecture*.

D.11.2 Refined Identically Zero Tensors: Fixed Elevations

```
tensorIdenticallyZeroQ[
      minSignedConfiguration_, signedConfigs_List, T■ : _List] :=
   Block[
      {allT, minConfiguration = minSignedConfiguration[[2]], inducingRules},
      inducingRules = Flatten @ Cases[minConfiguration, s(i_,j_) → {i → j}];
      allT = T■ ⋃ (inducedTransposition /@ T■);
      (Union[sc[[2]] &sc /@ signedConfigs]len < signedConfigslen) ⋁
         directlyZeroQ[minConfiguration, T■] ⋁
         zeroByComplementaryPairQ[minConfiguration, allT]]
```

`zeroByComplementaryPairQ` proceeds by removing the signs from the set $\mathcal{T}_{\blacksquare} \bigcup induced_a(\mathcal{T}_{\blacksquare})$ and finding any duplicates in the resulting set. If any of the index pairs in this set operate on indices which are not both fixed and of opposite elevations, then the tensor is zero.

```
zeroByComplementaryPairQ[minConfiguration_, allT_] :=
   Or @@ (permissibleZeroIndicesQ @ minConfiguration[[#]] &) /@
         Cases[Split @ Sort[st[[2]] &st /@ allT], {pair_, pair_} → pair]

permissibleZeroIndicesQ @ {s(↑,_), s(↓,_)} = False;
permissibleZeroIndicesQ @ {s(↓,_), s(↑,_)} = False;
permissibleZeroIndicesQ @ other_ = True;
```

D.12 Refinements for Varying Index Classes

D.12.1 Refined Transpositional Canonicalization Operator: Index Classes & Fixed Elevations

This code is predicated on §C.2 *Evidence for Steepest Descent Conjecture*. It is necessary if we use tensors containing mixed index types. It *exactly* follows the code for the first extension of the summed indices to indices with fixed elevations. The only initial change is to add a slot for index classes in our summed indices.

$$s^{class1\_}_{i\_,m\_} \leftrightarrows_? s^{class2\_}_{j\_,n\_} := \text{compareS}[i, j, m, n];$$

Clearly we should not swap $\uparrow \leftrightarrows \downarrow$ but we should swap $\downarrow \leftrightarrows \uparrow$.

$$s^{-,\uparrow}_{i\_,m\_} \leftrightarrows_? s^{-,\downarrow}_{j\_,n\_} = \text{False};$$
$$s^{-,\downarrow}_{i\_,m\_} \leftrightarrows_? s^{-,\uparrow}_{j\_,n\_} = \text{True};$$

Also if both signs are the same then we should clearly just order using the normal conventions.

$$s^{-,\uparrow}_{i\_,m\_} \leftrightarrows_? s^{-,\uparrow}_{j\_,n\_} = n < m;$$
$$s^{-,\downarrow}_{i\_,m\_} \leftrightarrows_? s^{-,\downarrow}_{j\_,n\_} = n < m;$$

And finally when we mix kinds, we adopt the following.

$$s^{-,\downarrow}_{i\_,m\_} \leftrightarrows_? s^{-}_{j\_,n\_} = m > n \vee n > i;$$
$$s^{-,\uparrow}_{i\_,m\_} \leftrightarrows_? s^{-}_{j\_,n\_} = j < n < m;$$
$$s^{-}_{i\_,m\_} \leftrightarrows_? s^{-,\downarrow}_{j\_,n\_} = n < m < i;$$
$$s^{-}_{i\_,m\_} \leftrightarrows_? s^{-,\uparrow}_{j\_,n\_} = j > m \vee m > n;$$

D.12.2 Complimentary-Effect Transpositional Pair Canonicalization

$$@_{S\_List}\left[\overline{configuration_{\_signC\_}}\right] /; optIndexClassInclusion :=$$
$$\text{Block}\Big[\{newConfiguration = configuration,$$
$$newSign = signC,\ partialConfig = \{\},\ \mathcal{T},\ \mathcal{N},\ \mathcal{T} = S\},$$
$$\text{While}[partialConfig \neq newConfiguration,$$
$$partialConfig = newConfiguration;$$
$$\text{transposeIfMoreCanonical } /@\ \mathcal{T}];$$
$$\text{complementaryTransposeIfMoreCanonical } /@$$
$$\text{Cases}\left[\mathcal{T},\ (i\_ \leftrightarrow j\_)_{signt\_} /; \text{mixesIndexClassesQ}@\ partialConfig_{[\![\{i,j\}]\!]}\right];$$
$$\text{While}[partialConfig \neq newConfiguration,$$
$$partialConfig = newConfiguration;$$
$$\text{complementaryTransposeIfMoreCanonical } /@\ \mathcal{T}];$$
$$\overline{partialConfig_{newSign}}\Big]$$

$$\text{mixesIndexClassesQ } @\ \{s^{class1\_?\text{ValidIndexClassQ},\_\_}_{\_,\_},\ s^{class2\_?\text{ValidIndexClassQ},\_\_}_{\_,\_}\} :=$$
$$class1 \neq class2$$
$$\text{mixesIndexClassesQ } @\ other\_ = \text{False};$$

$$\text{complementaryTransposeIfMoreCanonical } @\ (i\_ \leftrightarrow j\_)_{signt\_} :=$$
$$\text{reducingComplementarySwap}[signt,\ newConfiguration_{[\![\{i,j\}]\!]}]$$

$$\text{reducingComplementarySwap}\left[signt\_,\ \left\{ai : s^{class1\_?\text{ValidIndexClassQ},\_\_}_{i\_,m\_},\ aj : s^{class2\_?\text{ValidIndexClassQ},\_\_}_{j\_,n\_}\right\}\right] /;$$

```
        class2 <tex class1 ∧ i < j < Min[m, n] ∧
            doubleSwapIsMoreCanonicalQ[ai, aj] :=
    Block[{foundSign},
        foundSign = Cases[T, (Min[m, n] ↔ Max[m, n])sign_ → sign];
        If[foundSign ≠ {},
            doubleSwapThem[signT (First @ foundSign), i, j, m, n];]]

reducingComplementarySwap @ other__ = False;

doubleSwapIsMoreCanonicalQ[s(class1_)i_,m_, s(class2_)j_,n_] = True;

doubleSwapIsMoreCanonicalQ[s(class1_,↓)i_,m_, s(class2_,↓)j_,n_] = True;
doubleSwapIsMoreCanonicalQ[s(class1_,↑)i_,m_, s(class2_,↑)j_,n_] = True;

doubleSwapIsMoreCanonicalQ[s(class1_,↑)i_,m_, s(class2_)j_,n_] = True;
doubleSwapIsMoreCanonicalQ[s(class1_)i_,m_, s(class2_,↑)j_,n_] = True;

doubleSwapIsMoreCanonicalQ[_, _] = False;

doubleSwapThem[signTT_, i_, j_, m_, n_] := (
        {newConfiguration[[i]], newConfiguration[[j]], newConfiguration[[m]], newConfiguration[[n]]} =
            newConfiguration[[{j, i, n, m}]];
        newSign = newSign signTT;
        newConfiguration =
            Replace[newConfiguration, {i → j, j → i, n → m, m → n}, {2}];);
```

D.12.3 Refinement to tensorIdenticallyZeroQ

We must modify one final piece of code. The test to see whether a tensor is identically zero proceeds almost as before, except the test to determine if there exists a signed transposition $\tau \in \mathcal{T}_{\blacksquare}$ such that both τ and $-\tau$ are present in $\mathcal{T}_{\blacksquare} \bigcup induced_a(\mathcal{T}_{\blacksquare})$. We must also check that this transposition τ only affects indices of the same class. If it moves indices in different classes, then the configurations are distinguishable and thus the tensor is not identically zero.

Therefore, our code is very similar to the previous code, except that the final condition is changed in `zeroByComplementaryPairQ`.

```
permissibleZeroIndicesQ @ {s_,_↑, s_,_↓} = False;
permissibleZeroIndicesQ @ {s_,_↓, s_,_↑} = False;
permissibleZeroIndicesQ @
        {s(class1_?ValidIndexClassQ)_,__, s(class2_?ValidIndexClassQ)_,__} := class1 ≡ class2;
permissibleZeroIndicesQ @ other_ = True;
```

Finally, we need to slightly change one of the tests for `directlyZeroQ`. As long as we are not operating on a pair of indices with fixed elevations, then a tensor can still be directly zero.

```
Clear @ reduceIndex
```

```
reduceIndex @ s_[i_, j_] := s @ Min[i, j];
reduceIndex @ s_[i_, j_]^?ValidIndexClassQ := s @ Min[i, j];
reduceIndex @ h_[index_, ___] = h @ index;
```

D.12.4 Canonicalization Examples with Mixed Index Classes

Omitted.

D.13 The Complete Canonicalization Algorithm

D.13.1 SetUps

Canonicalize options.

```
Options @ Canonicalize = {
      MetricLocallyFlat → False,
      LinearSymmetryMethod → GröbnerBases,
      IndexClassInclusion → True};

extraEquationsOfTensorSymbol @ symb_ = True;

valueHeadQ @ expr_ @ body___ :=
   Hold @@ {expr @ body} ≠ Hold @ expr @ # & @@ { body}

SetAttributes[ valueHeadQ, HoldAll]
```

D.13.2 tensorProduct ↔ tensorSymbol

```
tensorProductToTensorSymbol @ tensorProduct_ :=
   With[{symb = Unique @ "TPS`Π"},
      tensorProductToTensorSymbol @ tensorProduct = symb;
      tensorSymbolToTensorProduct @ symb = tensorProduct;
      symb]

tensorProductSymbolQ @ symb_Symbol := Context @ symb === "TPS`"
```

D.13.3 signedConfiguration ↔ tensorSymbol

```
signedConfigurationToTensorSymbol[keySymbol_, config__] :=
   -signedConfigurationToTensorSymbol[keySymbol, config+];
signedConfigurationToTensorSymbol[keySymbol_, 0] = 0;

signedConfigurationToTensorSymbol[keySymbol_, signedConfig_] :=
   signedConfigurationToTensorSymbol[keySymbol, signedConfig] =
      Block[{tensorProduct, tensorSymbol},
         tensorSymbol = Unique @ "TPS`Π";
         tensorProduct = reconstituteTensorsCached[keySymbol, signedConfig];
         tensorSymbolToTensorProduct @ tensorSymbol = tensorProduct;
         tensorProductToTensorSymbol @ tensorProduct = tensorSymbol;
         canonicalSymbolOfTensorProduct @ tensorProduct = tensorSymbol;
         orderOfTensorSymbol @ tensorSymbol = keySymbol;
         tensorSymbolToSignedConfiguration @ tensorSymbol =
            signedConfig;
         tensorSymbol
      ];
```

D.13.4 reconstituteTensorsCached

This is a cached scheme to avoid the recalculation of tensors when we have already calculated the corresponding tensor product for a given signed configuration with a given tensor key, that is, knowing the names and valences of a tensor. Note: *tensorIdentifiers* is a dynamic variable that is set inside `canonicalizeTerm` and, as such, does not have to be explicitly passed in.

```
reconstituteTensorsCached[keySymbol_, signedConfig_] :=
   reconstituteTensorsCached[keySymbol, signedConfig] =
      reconstituteTensors[
         First @ keySymbolToKeyOrder @ keySymbol, tensorIdentifiers, signedConfig]
```

D.13.5 Clear Caches

We have just presented the individual code pieces that need to be reinitialized whenever we clear the tensor caches. In full, here is the code.

```
ClearTensorCaches[] :=
   (Clear[linearEquationsOfTensorSymbol,
         signedConfigurationToTensorSymbol,
         canonicalSymbolOfTensorProduct, orderOfTensorSymbol,
         tensorProductToTensorSymbol, reconstituteTensorsCached];

      reconstituteTensorsCached[keySymbol_, signedConfig_] :=
         reconstituteTensorsCached[keySymbol, signedConfig] =
            reconstituteTensors[First @ keySymbolToKeyOrder @ keySymbol,
```

```
        tensorIdentifiers, signedConfig];

    tensorProductToTensorSymbol @ tensorProduct_ :=
      With[{symb = Unique @ "TPS`Π"},
        tensorProductToTensorSymbol @ tensorProduct = symb;
        tensorSymbolToTensorProduct @ symb = tensorProduct;
        symb];

    signedConfigurationToTensorSymbol[keySymbol_, config__] :=
      -signedConfigurationToTensorSymbol[keySymbol, config+];
    signedConfigurationToTensorSymbol[keySymbol_, 0] = 0;

    signedConfigurationToTensorSymbol[keySymbol_, signedConfig_] :=
      signedConfigurationToTensorSymbol[keySymbol, signedConfig] =
        Block[{tensorProduct, tensorSymbol},
          tensorSymbol = Unique @ "TPS`Π";
          tensorProduct =
            reconstituteTensorsCached[keySymbol, signedConfig];
          tensorSymbolToTensorProduct @ tensorSymbol = tensorProduct;
          tensorProductToTensorSymbol @ tensorProduct = tensorSymbol;
          canonicalSymbolOfTensorProduct @ tensorProduct = tensorSymbol;
          orderOfTensorSymbol @ tensorSymbol = keySymbol;
          tensorSymbolToSignedConfiguration @ tensorSymbol =
            signedConfig;
          tensorSymbol
        ];)
```

D.13.6 keyOrder ↔ keySymbol

```
keyOrderToKeySymbol @ keyOrder_ :=
  With[{keySymbol = Unique @ "key"},
    keyOrderToKeySymbol @ keyOrder = keySymbol;
    keySymbolToKeyOrder @ keySymbol = keyOrder;
    keySymbol]
```

D.13.7 tensorExprs ↔ tensorSymbols

```
tensorExprsToTensorSymbols @ tensors_Plus :=
  tensorExprsToTensorSymbols /@ tensors;
tensorExprsToTensorSymbols [ a_ == b_ ] :=
  tensorExprsToTensorSymbols @ a == tensorExprsToTensorSymbols @ b;
tensorExprsToTensorSymbols [ nt_ tensorProduct_] :=
  nt tensorExprsToTensorSymbols @ tensorProduct /; FreeQ[nt, Tensor];
tensorExprsToTensorSymbols @ (Times @ tensorProduct__Tensor)_hp :=
  tensorProductToTensorSymbol @ Times @ tensorProduct;
tensorExprsToTensorSymbols @ tensor_Tensor :=
  tensorProductToTensorSymbol @ tensor;
tensorExprsToTensorSymbols @ other_ = other;
```

```
tensorSymbolsToTensorExprs @ expr_ := expr /.
      symb_?tensorProductSymbolQ → tensorSymbolToTensorProduct @ symb;
```

D.13.8 Canonicalize

```
Canonicalize [ expr_, options___?OptionQ] :=
   Block[{
         optMetricLocallyFlat =
            MetricLocallyFlat /. {options} /. Options[Canonicalize],
         optLinearSymmetryMethod = LinearSymmetryMethod /. {options} /.
               Options[Canonicalize],
         optIndexClassInclusion = IndexClassInclusion /. {options} /.
               Options[Canonicalize],
         optLinearSymmetries},
      optLinearSymmetries := ¬ (optLinearSymmetryMethod ≡ None);
      Which[
         optLinearSymmetries,      canonicalizeLinear @ expr,
         ¬ optLinearSymmetries,    canonicalizeNoLinear @ expr]]

canonicalizeNoLinear @ expr_ :=
   tensorSymbolsToTensorExprs @ canonicalizeTerm @ expr

canonicalizeLinear @ expr_ :=
   Block[{equations, tensorSymbolsInEquations, canonicalSymbols},
      canonicalSymbols = canonicalizeTerm @ expr;
      tensorSymbolsInEquations =
         connectedTensorSymbols @ Union @ Cases[{canonicalSymbols},
               symb_?tensorProductSymbolQ, {0, ∞}];
      equations = (Flatten [ linearEquationsOfTensorSymbol /@
                        tensorSymbolsInEquations] ⋃
                  Flatten [ extraEquationsOfTensorSymbol /@
                        tensorSymbolsInEquations]) \ {True};

      reduceExprWRTEquations[canonicalSymbols, optLinearSymmetryMethod]]
```

Both these functions rely on the implicit dynamic passing of the variables *tensorSymbolsInEquations* and *equations*. These variables are not "fundamental" to the method, so this choice was taken. (Really, it would be nice if *Mathematica* had local functions.)

```
reduceExprWRTEquations[canonicalSymbols_, DirectReduction] :=
   tensorSymbolsToTensorExprs [
      canonicalSymbols //. Flatten[
            With[{variables = Cases[{eqn}, _?tensorProductSymbolQ, {0, ∞}]}
                  Solve[eqn, Last @ sortTensorSymbols @
                        variables]] &eqn /@ equations]]
```

If we are not using the direct reduction method for simplifying the expression, then we are using the Gröbner-bases method.

```
reduceExprWRTEquations[canonicalSymbols_, (GröbnerBases | _)] :=
   Block[{orderedTensorSymbols = Reverse @
```

```
                sortTensorSymbols @ tensorSymbolsInEquations, groebnerBasis},
      groebnerBasis := GroebnerBasis[equations, orderedTensorSymbols];
      tensorSymbolsToTensorExprs @ reduceExpr @ canonicalSymbols]

reduceExpr @ expr_Equal := reduceExpr @ term &term /@ expr;
reduceExpr @ expr_ /; PolynomialQ[expr, orderedTensorSymbols] :=
   Last @ PolynomialReduce[expr, groebnerBasis, orderedTensorSymbols];
reduceExpr @ expr_Equal @ args__ := (reduceExpr @ term &term ) /@ expr;
reduceExpr @ other___ = other;
```

D.13.9 AdditionalEquations

```
Options @ AddAuxiliaryEquations = {
      MetricLocallyFlat → False,
      IndexClassInclusion → True};

AddAuxiliaryEquations[ eqn_Equal, options___?OptionQ] :=
   AddAuxiliaryEquations [ {eqn}, options]

AddAuxiliaryEquations[ {eqns__Equal}, options___?OptionQ] :=
   Block[{
         optMetricLocallyFlat = MetricLocallyFlat /. {options} /.
               Options[AddAuxiliaryEquations],
         optIndexClassInclusion = IndexClassInclusion /. {options} /.
               Options[AddAuxiliaryEquations],
         optLinearSymmetries = True,
         equations, symblifiedExpr, tensorSymbolsInEquations,
         orderedTensorSymbols, canonicalSymbolEqns},

      canonicalSymbolEqns = canonicalizeTerm /@ {eqns};
      storeEquations /@ canonicalSymbolEqns;
   ]

storeEquations @ eqn_ := storeEquationForSymbol[eqn, #] & /@
      Union @ Cases[{eqn}, _?tensorProductSymbolQ, {0, ∞}]

storeEquationForSymbol[eqn_, symb_?tensorProductSymbolQ] :=
   If[extraEquationsOfTensorSymbol @ symb == True,
      extraEquationsOfTensorSymbol @ symb = {eqn},
      extraEquationsOfTensorSymbol @ symb =
         (extraEquations @ symb) ⋃ {eqn}]
```

D.13.10 EquationsOfExpression

```
Options @ EquationsOfExpression = {
      ReturnOptimizedEquations → False,
      MetricLocallyFlat → False,
      IndexClassInclusion → True};

EquationsOfExpression [ expr_, options___?OptionQ] :=
   Block[{
         optMetricLocallyFlat = MetricLocallyFlat /. {options} /.
```

```
            Options[EquationsOfExpression],
        optIndexClassInclusion = IndexClassInclusion /. {options} /.
            Options[EquationsOfExpression],
        optReturnOptimizedEquations = ReturnOptimizedEquations /. {options} /.
            Options[EquationsOfExpression],
        optLinearSymmetries = True,
        equations, tensorSymbolsInEquations, orderedTensorSymbols, canonicalSymbols},

    canonicalSymbols = canonicalizeTerm @ expr;
    tensorSymbolsInEquations =
        connectedTensorSymbols @ Union @ Cases[{canonicalSymbols},
            symb_?tensorProductSymbolQ, {0, ∞}];
    equations = (Flatten[linearEquationsOfTensorSymbol /@
            tensorSymbolsInEquations] ⋃
        Flatten[extraEquationsOfTensorSymbol /@
            tensorSymbolsInEquations]) \ {True};
    orderedTensorSymbols = Reverse @ sortTensorSymbols @
        tensorSymbolsInEquations;
    tensorSymbolsToTensorExprs @ If[optReturnOptimizedEquations,
        (0 == # &) /@ GroebnerBasis[equations, orderedTensorSymbols],
        equations]]
```

D.13.11 sortTensorSymbols

```
sortTensorSymbols @ symbols_ :=
    Sort[symbols, totalOrderOnTensorSymbols[#1, #2] &]

totalOrderOnTensorSymbols[symbol1_, symbol2_] :=
    Block[{ordering},
        ordering = Order[orderOfTensorSymbol @ symbol1,
            orderOfTensorSymbol @ symbol2];
        If[ordering == 0, ordering =
            tensorSymbolToSignedConfiguration @ symbol1 ≶c
                tensorSymbolToSignedConfiguration @ symbol2];
        If[ordering ≤ 0, False, True]]
```

D.13.12 connectedTensorSymbols

`connectedTensorSymbols` works in almost the exact same manner as the algorithm for generating configurations. We start with some seed symbols. We then find all the equations involving these seed symbols. If we find any new symbols in these equations, then add these symbols to the found symbols, and so on. We only stop when we find no new symbols in the latest equations. In this way we will find all relevent symbols that are joined together by equations.

```
connectedTensorSymbols @ {} = {};

connectedTensorSymbols @ {seedSymbols__Symbol} :=
    Block[{newEquations = {}, ℱ = {}, 𝒩 = {seedSymbols}},
        While[𝒩 ≠ {},
```

```
    F = F ⋃ N;
    newEquations = (extraEquationsOfTensorSymbol /@ N) ⋃
          (linearEquationsOfTensorSymbol /@ N);
    N = Cases[newEquations, symb_?tensorProductSymbolQ, {0, ∞}] \ F];
  F]
```

D.13.13 canonicalizeTerm

`canonicalizeTerm` takes an expression containing tensors and returns an expression with all the tensors replaced by symbols referring to canonical versions of the tensors. We can strip out any multipliers and distribute over any sums.

```
canonicalizeTerm @ tensors_Plus := canonicalizeTerm /@ tensors;
canonicalizeTerm @ expr : head_[___, sum_Plus, ___] :=
   canonicalizeTerm @ Distribute @ expr /;
      tensorialMultiplicativeHeadQ @ head
canonicalizeTerm [ nt_ tensorProduct_] :=
   nt canonicalizeTerm @ tensorProduct /; FreeQ[nt, Tensor];
canonicalizeTerm [ ( nt : Tensor[name_, {}]^(m_.) )_hp tensorProduct_] :=
   nt canonicalizeTerm @ tensorProduct;
canonicalizeTerm @ eqn_Equal := canonicalizeTerm /@ eqn;
canonicalizeTerm @ other_ = other;
```

If the tensor product has already been canonicalized and linear symmetries are being used, then we return the cached value, if there is one and if the equations have already been generated.

```
canonicalizeTerm @ tensorProduct : (head_ @ __Tensor | _Tensor) :=
   canonicalSymbolOfTensorProduct @ tensorProduct /;
      optLinearSymmetries ∧
         (ValueQ @ canonicalSymbolOfTensorProduct @ tensorProduct) ∧
          (valueHeadQ @ linearEquationsOfTensorSymbol @ normalize @
                    canonicalSymbolOfTensorProduct @ tensorProduct)

normalize[nt_?NumberQ tensorProduct_] = tensorProduct;
normalize @ tensorProduct_ = tensorProduct;
```

If the tensor product has previously been canonicalized and we are not including linear symmetries, then we return the cached value, if there is one.

```
canonicalizeTerm @ tensorProduct : (head_ @ __Tensor | _Tensor) :=
   canonicalSymbolOfTensorProduct @ tensorProduct /;
      ¬ optLinearSymmetries ∧
         (ValueQ @ canonicalSymbolOfTensorProduct @ tensorProduct)
```

In all other cases we must actually perform the computation. We encode the tensor and calculate the generators of its signed symmetry group. Then we canonicalize the free indices in the signed configuration. Then we pass it off to the canonicalizing engine. And finally, we calculate the governing equations of the linear symmetries.

```
canonicalizeTerm @ tensorProduct : (head_ @ __Tensor | _Tensor) :=
   Block[{S_T, S_X, S_D, S_L, (S_T)_■, productHead, tensorIdentifiers, signedConfiguration,
         tensorId, configurationLength, minSignedConfiguration, keySymbol, tensorSymbol,
```

```
    canonicalizeEngine, equationsGeneratedQ, options},

(* we must include the options sent dynamically from
        functions which call this one when encoding the tensor product *)
options = Sequence[MetricLocallyFlat → optMetricLocallyFlat,
        IndexClassInclusion → optIndexClassInclusion];
{productHead, tensorIdentifiers, signedConfiguration} =
    encodeTensors [tensorProduct, options];
keySymbol = keyOrderToKeySymbol @
        Prepend[key @@ (T[order] &T) /@ tensorIdentifiers, productHead];

(* calculate the generators to be used *)
configurationLength = signedConfiguration[[2]]len;
{S𝒯, S𝒳, S𝒟, Sℒ, (S𝒯)■} =
    calculateAllGenerators @ tensorIdentifiers;

(* find the minimum equivalent signed configuration, caching values as we progress *)
canonicalizeEngine @ signedConfig_ :=
    canonicalizeEngine @ signedConfig =
        canonicalizeEngineAux @ signedConfig;
minSignedConfiguration = canonicalizeEngine @ signedConfiguration;

tensorSymbol = signedConfigurationToTensorSymbol [
        keySymbol, minSignedConfiguration];

(* if we have not already calculated all other signed
        configurations reachable by the linear symmetries then do so *)
If[optLinearSymmetries ∧ ¬ valueHeadQ @
            linearEquationsOfTensorSymbol @
                normalize @ tensorSymbol,
    generateEquations @ minSignedConfiguration];

canonicalSymbolOfTensorProduct @ tensorProduct = tensorSymbol]
```

Of course, many different variations on the total algorithm are possible. We could refrain from generating any new linear symmetry equations and instead just use the existing linear symmetries. Alternatively, we could ignore linear symmetries altogether. However, we must ensure that the options do not conflict with one another.

D.13.14 canonicalizeEngine

`canonicalizeEngine` takes a signed configuration and returns the corresponding equivalent signed configuration that is canonical with respect to all symmetries except the linear symmetries. It is a cached set of values that is cleared for each term.

Canonicalizing a negative configuration yields the opposite configuration of the canonicalized positive configuration.

```
canonicalizeEngineAux @ config-_ :=
    oppositeConfiguration @ canonicalizeEngine @ config+;
```

```
oppositeConfiguration @ config__sign_ = config_-sign;
oppositeConfiguration @ 0 = 0;
```

If we try to canonicalize a configuration where the free indices are not in their canonical position, then move the free indices to their canonical positions and try again.

```
freeIndicesAreCanonical = False;

canonicalizeEngineAux @ signedConfig_ /; ¬ freeIndicesAreCanonical :=
    Block[{freeIndicesAreCanonical, answer},
        answer = canonicalizeFreeIndices[signedConfig, S_T, S_X, S_D];
        freeIndicesAreCanonical = True;
        canonicalizeEngine @ answer]
```

Finally, if all else has failed, we actually have to do the calculation. This finds the minimum equivalent signed configuration.

```
canonicalizeEngineAux @ signedConfiguration_ :=
    Block[{S'_D, S'_X, S'_T, minSignedConfiguration, allEquivalentConfigurations},
        {S'_T, S'_X, S'_D} = stabilizePermutations[signedConfiguration, S_T, S_X, S_D];
        allEquivalentConfigurations =
            generateConfigurations_S'_T [signedConfiguration, S'_D ⋃ S'_X];
        minSignedConfiguration = Min_c @ allEquivalentConfigurations;
        If[tensorIdenticallyZeroQ[
                minSignedConfiguration, allEquivalentConfigurations, (S_T)_■], 0,
            canonicalizeEngine @ absSignedConfig @ minSignedConfiguration =
                absSignedConfig @ minSignedConfiguration;
            minSignedConfiguration]]

absSignedConfig @ config__ = config_+;
absSignedConfig @ expr_ = expr;
```

D.13.15 generateEquations

The equations generated are independant of the sign of the signed configuration, and in addition we don't need to generate any equations for a zero tensor.

```
generateEquations @ 0 = True;
generateEquations @ config__ := generateEquations @ config_+
generateEquations @ signedConfig_ :=
    Null /; equationsGeneratedQ @ signedConfig
```

Given a signed configuration that is canonical with respect to everything except linear symmetries, we build all the equations dictated by the linear symmetries. We then express the tensors in these equations as tensor symbols. Finally, we put these equations into a standard form using Reduce and then save them in the appropriate place.

```
generateEquations @ signedConfig_signedConfiguration :=
    Block[{tensorSymbol, equations},
        tensorSymbol =
            signedConfigurationToTensorSymbol[keySymbol, signedConfig];
```

```
    equationsGeneratedQ @ signedConfig = True;
    equations = Flatten [buildEquation[sl, signedConfig] &sl /@ SL];
    generateEquations /@
       Flatten @ Cases[equations, _signedConfiguration, {0, ∞}];
    equations = equations /. sc_signedConfiguration :↦
              signedConfigurationToTensorSymbol[keySymbol, sc] ;
    equations = Union @ Flatten @
              {( equations /. eqn_Equal :↦ Reduce[eqn]) /. And → List};
    If[Head @ linearEquationsOfTensorSymbol @ tensorSymbol ≠ List ,
       linearEquationsOfTensorSymbol @ tensorSymbol = equations];]
```

D.13.16 buildEquation

`buildEquation` just builds an equation corresponding to a linear symmetry. It takes as arguments a linear symmetry and a signed configuration and returns an equation containing signed configurations that are canonical with respect to all the symmetries except the linear symmetries.

```
buildEquation[LinearSymmetry @ perms__, sc_signedConfiguration] :=
   0 == Plus @@ (canonicalizeEngine[sp *c sc] &sp /@ {perms})

On[General::"spell1"]
```

D.14 User Functions and InputAliases

D.14.1 User Functions for Symmetries

We define a special complex notation in order to circumvent the parsing of the requested generator specifications. That is, we avoid introducing the symbols $\mathcal{T}$, $\mathcal{X}$, $\mathcal{D}$, $\mathcal{L}$ and $\mathcal{T}_{\blacksquare}$. This also avoids any problems due to package contexts, etc.

```
Notation[⟨tensorProduct_⟩S ⟺ StandardSymmetries[tensorProduct_]];
Notation[⟨tensorProduct_⟩Stype__ ⟹
      Symmetries[tensorProduct_, Tensors`Private`literalBoxes[type__]]];
Tensors`Private`literalBoxes /:
   MakeExpression[any_Tensors`Private`literalBoxes, _] :=
   HoldComplete @ validSymmetryTypes @ any;
validSymmetryTypes @ types_ :=
   Cases[types, "𝒯" | "𝒳" | "𝒟" | "ℒ" | SubscriptBox["𝒯", "■"]];
```

```
StandardSymmetries[tensorProduct_] :=
   Flatten @ Symmetries[tensorProduct, {"𝒯", "𝒳", "𝒟"}]

SetAttributes[
   {StandardSymmetries, Symmetries, symmetriesAux}, HoldFirst]

Symmetries[tensorProduct : (head_ @ __Tensor)_hp, types_List] :=
   symmetriesAux[tensorProduct, types]
Symmetries[tensor_Tensor_hp, types_List] := symmetriesAux[{tensor}, types]

symmetriesAux[tensorProduct_, types_List] :=
   Block[
      {productHead, tensorIdentifiers, signedConfiguration, configurationLength, parts},
      {productHead, tensorIdentifiers, signedConfiguration} =
         encodeTensors[tensorProduct, sortTensors → False];
      configurationLength = signedConfiguration[[2]]_len;
      parts =
         Sequence @@ (Hold @ types /. {"𝒯" → 1, "𝒳" → 2, "𝒟" → 3, "ℒ" → 4,
                        SubscriptBox["𝒯", "■"] → 5});
      (calculateAllGenerators @ tensorIdentifiers)[[parts]]]
```

Remembering that the signed configuration is the third component returned by `encodeTensors`,

```
Notation[⟨tensorProduct_⟩_c ⟺ toSignedConfiguration[tensorProduct_]]

⟨head_ @ tensors__Tensor⟩_c :=
   encodeTensors[{tensors}, sortTensors → False]_[[3]]
⟨tensor_Tensor⟩_c := encodeTensors[{tensor}, sortTensors → False]_[[3]]

SetAttributes[toSignedConfiguration, HoldAll]
```

D.14.2 Input Aliases

Omitted.

D.15 Index Handling and Reindexing

`UsedIndices[`*expr*`]`	returns the indices used in the expression *expr*.
`ReIndex[`*expr*`]`	reindex all dummy indices in the expression *expr* to canonical indices
`ExpandContraction[`*expr*`]`	expand out any contracted dummy indices into a sum of terms where the dummy indices range over their allowed coordinates
`Dummify[`*expr*`]`	reindex all dummy indices in the expression *expr*, into unique dummy indices which have not previously been used
`DummyIndices[`*expr*`]`	returns the list of dummy indices occurring in the expression *expr*

The public index handling functions.

D.15.1 Miscellaneous Functions

Next, we want the list of α which are neither numbers nor strings, but are such that α^+ or α^- is used in the expression *expr*.

```
UsedIndices @ expr_ :=
   Cases[expr, ((α_?IndexQ)^+ | (α_?IndexQ)^-) → α, {0, ∞}]

selectMultipleOccurrences @ list_ :=
   Cases[Split @ Sort @ list, {α_, __} → α]
selectDisjointMultipleOccurrences @ expr_ :=
    Union @ selectMultipleOccurrences @
          Flatten[(Union @ UsedIndices @ #) & /@ List @@ expr]

freeOfHolds @ f_Symbol :=
   FreeQ[Attributes @ f, HoldAll | HoldFirst | HoldRest |
         HoldAllComplete | HoldComplete | SequenceHold]

headIsTensorialProductHeadQ @ head_ @ expr___ :=
   tensorialMultiplicativeHeadQ @ head_u
SetAttributes[headIsTensorialProductHeadQ, HoldAll];
```

D.15.2 ReIndex

```
ReIndex @ expr_Plus := ReIndex /@ expr
ReIndex @ expr_Tensor := reLabel[expr, DummyIndices @ expr]
ReIndex@ expr_?headIsTensorialProductHeadQ :=
   reLabel[expr, DummyIndices @ expr]
ReIndex @ other_ = other;
```

We could have avoided this and indeed made our reindexing reduce things further by being clever with the implementation of `ReIndex`. However, the slowdown implicit in such an approach, coupled with the fact that `ReIndex` will almost always be called on expanded sums, leads us to conclude that it is not worthwhile to implement this extended `ReIndex`.

D.15.3 Dummify

```
Dummify @ expr_Plus := Dummify /@ expr
Dummify @ expr_Tensor :=
   reLabelUnique[expr, selectMultipleOccurrences @ UsedIndices @ expr]
Dummify@ expr_?headIsTensorialProductHeadQ := Dummify /@
      reLabelUnique[expr, selectDisjointMultipleOccurrences @ expr]
Dummify@ function_Symbol?freeOfHolds [args__] := function @@ Dummify /@ {args}
Dummify @ other_ = other;

reLabelUnique[expr_, indices_] :=
    With[
      {intermediateReplacements = Inner[Rule, indices, Unique @ ToString @ First @
                                    BaseIndices @ ClassOfIndex @ # & /@
                    indices, List]}, expr /. intermediateReplacements]
```

D.15.4 ExpandContraction

```
ExpandContraction @ expr_Plus := ExpandContraction /@ expr
ExpandContraction @ expr_Tensor :=
   sumOverIndices[expr, selectMultipleOccurrences @ UsedIndices @ expr]
ExpandContraction @ expr_?headIsTensorialProductHeadQ :=
   ExpandContraction /@
      sumOverIndices[expr, selectDisjointMultipleOccurrences @ expr]
ExpandContraction @ function_Symbol?freeOfHolds [args__] :=
   function @@ ExpandContraction /@ {args}
ExpandContraction @ other_ = other;
```

D.15.5 DummyIndices

```
DummyIndices @ expr_Plus :=
   Union @ Flatten @ Join[DummyIndices /@ List @@ expr]
DummyIndices @ expr_Tensor :=
   selectMultipleOccurrences @ UsedIndices @ expr
DummyIndices@ expr_?headIsTensorialProductHeadQ :=
   Union @ Flatten @ Join[selectDisjointMultipleOccurrences @ expr,
            DummyIndices /@ List @@ expr]
DummyIndices@ function_Symbol?freeOfHolds [args___] :=
   Union @ Flatten @ Join[ DummyIndices /@ {args}]
DummyIndices @ other_ = {};
```

D.15.6 reLabel

reLabel just relabels the given indices occurring in an expression to unique canonical indices.

```
reLabel[expr_, indices_] :=
    With[{intermediateIndices = Table[Unique @ "λ", {indices_fen}],
          newIndicesList = generateNewIndices[
                Union @ UsedIndices @ expr \ indices, indices]}, With[
          {finalReplacements = MapThread[Rule, {intermediateIndices, newIndicesList}],
             intermediateReplacements = MapThread[Rule, {indices, intermediateIndices}]},
          expr /. intermediateReplacements /. finalReplacements]]

generateNewIndices[indicesAlreadyUsed_, indicesToReplace_] :=
   Block[{indexTable},

      (* initialize index table *)
      indexTable @ usedIndices = indicesAlreadyUsed;

      (* generate new indices *)
      getIndex @ ClassOfIndex @ # & /@ indicesToReplace]
```

D.15.7 sumOverIndices

```
transformIterators[iterationAssignments___dummySet] :=
   Apply[Set, Hold @ {iterationAssignments}, {2}]

sumOverIndices[expr_, {}] = expr;
sumOverIndices[expr_, indices_] :=
   Plus @@ (Block[iterators, expr] &_iterators^HoldAll) @@@
         Flatten @
            (Outer[transformIterators, Sequence @@ iterators] &_iterators) @
               ((First @ Outer[dummySet, {index},
                                 CoordinatesOfIndexClass @
                                 ClassOfIndex @ index] &_index) /@ indices)
```

D.16 Package Endings

D.16.1 Add Styles and Aliases to the Notebook if Necessary

Omitted.

D.16.2 End The Package

```
friendlyOn @ General::spell1;

End[];

SetAttributes[
   {AddAuxiliaryEquations, BasicCanonicalize, Canonicalize,
      CanonicalizeBrief, ClearTensorCaches, Commutative,
      DeclareCoordinateClass, DeclareCoordinates, DeclareIndexClass,
      DeclareSymmetries, DeclareTensorialMultiplicativeHead,
      DirectReduction, Dummify, DummyIndices, EquationsOfExpression,
      ExpandContraction, GeneralIndices, GröbnerBases,
      IndeterminateIndexClass, IndexClassInclusion, LinearSymmetry,
      LinearSymmetryMethod, MetricLocallyFlat, OptimizedCanonicalize,
      ReIndex, ReturnOptimizedEquations, SignedPermutation,
      SignedTransposition, SpaceTimeIndices, StandardSymmetries,
      Symmetries, UsedIndices} , {ReadProtected, Protected}]

EndPackage[];
```

References

[1] ▪ Abadi M. and Cardelli L. (1996) *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag.

[2] ▪ Abbott J.A., van Leeuwen A. and Strotmann A. (1996) *Tr 12: Objectives of OpenMath*. Report RIACS. TU, Eindhoven.

[3] ▪ Abbott J.A., van Leeuwen A. and Strotmann A. (1998) *OpenMath Communicating Information between Co-operating Agents in a Knowledge Network*. Report http://www.brunel.ac.uk/~hssrjis/issue/index8.html. Brunel University.

[4] ▪ Abdali S.K., Cherry G.W. and Soiffer N. (1986) *An Object-Oriented Approach to Algebra System Design*. In the Proceedings of SYMSAC'86: ACM-SIGSAM Symposium on Symbolic and Algebraic Computation, Waterloo, Canada, 21-23 July, edited by B.W. Char. pp. 24-30. ACM Press.

[5] ▪ Adams W. and Lousraunau P. (1994) *An Introduction to Gröbner Bases*. Graduate Studies in Mathematics, vol. 3. AMS, Oxford University Press.

[6] ▪ Aho A.V., Sethi R. and Ullman J.D. (1986) *Compilers, Principles, Techniques, and Tools*. Addison-Wesley.

[7] ▪ Åman J. (1978) *Classi*, a computer program. Royal Institute of Technology, Stockholm.

[8] ▪ Amrhein B., Gloor O. and Küchlin W. (1996) *Walking Faster*. In the Proceedings of DISCO'96: Design and Implementation of Symbolic Computation Systems, Karlsruhe, Germany, Sept. 18-20, edited by J. Calmet and C. Limongelli. Lecture Notes in Computer Science **1128**, pp. 150-161. Springer-Verlag.

[9] ▪ Apel J. and Klaus U. (1997) *FELIX (for commutative polynomial rings and modules and non-commutative structures)*, version 3.0, a computer program. University of Leipzig. http://felix.hgb-leipzig.de/

[10] ▪ Arnon D., Beach R., *et al.* (1988) *CaminoReal: An Interactive Mathematical Notebook*. In the Proceedings of EP '88, an International Conference on Electronic Publishing, Document Manipulation, and Typography, Nice, France, edited by J.C. van Vliet. pp. 1-18. Cambridge University Press.

[11] ▪ Arnon D., Waldspurger C. and McIsaac K. (1987) *CaminoReal User Manual Version 1.0*. Report CSL-87-5. Xerox PARC.

[12] ▪ Ashcroft N.W., Mermin N.D. and Mermin D. (1976) *Solid State Physics*. Holt Reinhart & Winston.

[13] ▪ Azmoodeh M. (1990) *Abstract Data Types and Algorithms*, 2nd edition. Macmillan Computer Science Series. Macmillan.

[14] ▪ Backhouse R., Jansson P., *et al.* (1999) *Generic Programming - An Introduction*. In Advanced Functional Programming, edited by S.D. Swierstra, P.R. Henriques and J.N. Oliveira. Lecture Notes in Computer Science **1608**, pp. 28-115. Springer-Verlag.

[15] ▪ Balfagón A. and Jaén X. (1998) *TTC: Symbolic Tensor Calculus with Indices*. Computers in Physics **12**(3) pp. 286-289.

[16] ▪ Baumann G. (1995) *Mathematica in Theoretical Physics: Selected Examples from Classical Mechanics to Fractals*. TELOS/Springer-Verlag.

[17] ▪ Bayer D. and Stillman M. (1986) *The Design of Macaulay: A System for Computing in Algebraic Geometry and Commutative Algebra*. In the Proceedings of SYMSAC'86: ACM-SIGSAM Symposium on Symbolic and Algebraic Computation, Waterloo, Canada, 21-23 July, edited by B.W. Char. pp. 157-162. ACM Press.

[18] ▪ Beaudouin-Lafon M. (1994) *Object-oriented Languages: Basic Principles and Programming Techniques*. Chapman & Hall.

[19] ▪ Becker T. and Weispfenning V. (1993) *Gröbner Bases: A Computational Approach to Commutative Algebra*. Graduate Texts in Mathematics, vol. 141. Springer-Verlag.

[20] ▪ Bennett J.P. (1990) *Introduction to Compiling Techniques - a first Course Using Ansi C, Lex and Yacc*. McGraw-Hill.

[21] ▪ Biggs N. (1993) *Discrete Mathematics*. Clarendon Press.

[22] ▪ Bird R., Scruggs T.E. and Mastropieri M.A. (1998) *Introduction to Functional Programming Using Haskell*, 2nd edition. Prentice Hall Series in Computer Science. Prentice Hall.

[23] ▪ Bjorken J.D. and Drell S.D. (1964) *Relativistic Quantum Mechanics*. International Series in Pure and Applied Physics. McGraw-Hill.

[24] ▪ Blaschek G. (1994) *Object-oriented Programming with Prototypes*. Springer-Verlag.

[25] ▪ Bonadio A. (1989) *Theorist*, a computer program for Macintosh. Prescience Corp., San Francisco.

[26] ▪ Bosma W., Cannon J. and Matthews G. (1994) *Programming with Algebraic Structures: Design of the Magma Language*. In the Proceedings of ISSAC '94: International Symposium on Symbolic and Algebraic Computation, Oxford, UK, 20-22 July, edited by J. von zur Gathen and M. Giesbrecht. pp. 52-57. ACM Press.

[27] ▪ Bosma W., Cannon J. and Playoust C. (1997) *The Magma Algebra System I: The User Language*. Journal of Symbolic Computation **24**(3–4) pp. 235-265.

[28] ▪ Boullier P. (1994) *Dynamic Grammars and Semantic Analysis*. Report 2322. Institute National de Recherche en Informatique et en Automatique (INRIA).

[29] ▪ Brans C.H. (1995) *Computer Algebra and General Relativity*. In: Computer Algebra in Science and Engineering, edited by J. Fleischer, J. Grabmeier, F.W. Hehl*et al*, pp. 183-195. World Scientific Publishing.

[30] ▪ Brink D.M. and Satchler G.R., eds. (1994) *Angular Momentum*. Oxford University Press.

[31] ▪ Broadbery P. (1999) *First Course in Aldor*. NAG, Oxford, http://www.nag.co.uk/symbolic/AX/doc/course.ps.gz.

[32] ▪ Broadbery P., Gomez-Diaz T. and Watt S. (1995) *On the Implementation of Dynamic Evaluation*. In the Proceedings of ISSAC '95: International Symposium on Symbolic and Algebraic Computation, Montreal, July 10-12, edited by A.H.M. Levelt. pp. 77-84. ACM Press.

[33] ▪ Brücher L., Franzkowski J. and Kreimer D. (1998) *xloops - Automated Feynman Diagram Calculation*. Computer Physics Communications **115** p 140–160.

[34] ▪ Buchberger B. (1965) *An Algorithm for Finding a Basis for a Residue Class Ring of a Zero-Dimensional Polynomial Ideal*. Doctoral Thesis, University of Innsbruck, Austria.

[35] ▪ Buchberger B. (1976) *Some Properties of Gröbner-Bases for Polynomial Ideals*. ACM SIGSAM Bulletin **10**(4) pp. 19-24.

[36] ▪ Buchberger B. and Loos R. (1983) *Algebraic Simplification*. In: Computer Algebra - Symbolic and Algebraic Computation, 2nd edition, edited by B. Buchberger, G. Collins and R. Loos, pp. 11-43. Springer-Verlag.

[37] ▪ Buchberger B. and Winkler F., eds. (1998) *Gröbner Bases and Applications*. London Mathematical Society Lecture Note Series **251**. Cambridge University Press.

[38] ▪ Budd T. (1991) *An Introduction to Object-Oriented Programming*. Addison-Wesley.

[39] ▪ Budden F.J. (1972) *The Fascination of Groups*. Cambridge University Press.

[40] ▪ Burshteyn B. (1990) *Generation and Recognition of Formal Languages by Modifiable Grammars*. ACM SIGPLAN Notices **25**(12) pp. 45-53.

[41] ▪ Burshteyn B. (1990) *On the Modification of the Formal Grammar at Parse Time*. ACM SIGPLAN Notices **25**(5) pp. 117-123.

[42] ▪ Butler G. (1991) *Fundamental Algorithms for Permutation Groups*. Lecture Notes in Computer Science **559**. Springer-Verlag.

[43] ▪ Butler G. and Cannon J. (1988) *Cayley, Version 4: the User Language*. In the Proceedings of ISSAC '88: International Symposium on Symbolic and Algebraic Computation, Rome, July 4-8, edited by P. Gianni. Lecture Notes in Computer Science **358**, pp. 456-466. Springer-Verlag.

[44] ▪ Butler G. and Cannon J. (1990) *The Design of Cayley - a Language for Modern Algebra*. In the Proceedings of DISCO '90: Design and Implementation of Symbolic Computation Systems, Capri, Italy, April 10-12, edited by A. Miola. Lecture Notes in Computer Science **429**, pp. 10-19. Springer-Verlag.

[45] ▪ Butler P.H. (1981) *Point Group Symmetry Applications: Methods and Tables*. Plenum Press.

[46] ▪ Cabasino B., Paolucci P.S. and Todesco G.M. (1992) *Dynamic Parsers and Evolving Grammars*. ACM SIGPLAN Notices **27**(11) pp. 39-48.

[47] ▪ Cajori F. and Cajori G. (1993) *A History of Mathematical Notations. Volume 1: Notations in Elementary Mathematics; Volume 2: Notations Mainly in Higher Mathematics*. Dover Publishing.

[48] ▪ Calmet J. and Limongelli C., eds. (1996) *International Symposium DISCO'96*. Lecture Notes in Computer Science **1128**. Springer-Verlag.

[49] ▪ Cannam C. (1993) *REDUCE XR: User Interface under X*. Konrad-Zuse-Zentrum, Berlin.

[50] ▪ Cannon J. and Playoust C. (1996) *MAGMA: A New Computer Algebra System*. Euromath-Bulletin **2**(1) pp. 113-144.

[51] ▪ Cannon J.J. (1982) *An Introduction to the Group Theory Language, Cayley*. In Proceedings of the London Mathematical Society Symposium on Computational Group Theory, Durham, UK, edited by M.D. Atkinson. pp. 145-183. Academic Press.

[52] ▪ Capracce H. (1995) *CANTENS (for manipulations and simplifications of indexed objects)*, a computer package for REDUCE, 3.7. Institute de Physique, Liege, Belgium.

[53] ▪ Cardelli L. (1988) *A Semantics of Multiple Inheritance*. Information and Computation **76** pp. 138-164.

[54] ▪ Cardelli L. and Wegner P. (1985) *On Understanding Types, Data Abstraction, and Polymorphism*. ACM Computing Surveys **17**(4) pp. 471-522.

[55] ▪ Cassels J.M. (1995) *Basic Quantum Mechanics*, 2nd edition. Krieger Publishing.

[56] ▪ Castellvi P., Jaén X. and Llanta E. (1994) *TTC: Symbolic Tensor and Exterior Calculus*. Computers in Physics **8**(3) pp. 360-367.

[57] ▪ Char B.W., Fee G.J., *et al.* (1995) *First Leaves: Tutorial Introduction to Maple V*. Springer-Verlag.

[58] ▪ Char B.W., Geddes K.O., *et al.* (1991) *Maple V Language Reference Manual*. Springer-Verlag.

[59] ▪ Char B.W., Geddes K.O., *et al.* (1991) *Maple V Library Reference Manual*. Springer-Verlag.

[60] ▪ Christensen S.M. (1998) *Large Scale Tensor Analysis by Computer*. Computer Physics Communications **115**(2-3) pp. 245-263.

[61] ▪ Christiansen H. (1990) *A Survey of Adaptable Grammars*. ACM SIGPLAN Notices **25**(11) pp. 35-44.

[62] ▪ Chu K.C.-K., Sarell C.E., *et al.* (1996) *General Relativity Calculations in Maple.* In the Proceedings of the 6th Canadian Conference in General Relativity and Relativistic Astrophysics, Fredericton, NB, May 25-27, edited by S.P. Braham, J.D. Gegenberg and R.J. McKellar. pp. 195-199. American Mathematical Society.

[63] ▪ Chyzak F. (1998) *Mgfun (among other aspects, deals with algebras of linear algebras)*, a computer package for Maple. INRIA.

[64] ▪ Cioni G., Colagrossi A. and Temperini M. (1996) *An Approach to Class Reasoning in Symbolic Computation.* Istituto di Analisi dei Sistemi ed Informatica, IASI-CNR.

[65] ▪ Cohen A.M., ed. (1993) *Computer Algebra in Industry - Problem Solving in Practice.* John Wiley.

[66] ▪ Cohen A.M., Lunel S.M. and Van Gastel L., eds. (1995) *Computer Algebra in Industry 2 - Problem Solving in Practice.* John Wiley.

[67] ▪ Cohen I., Frick I. and Åman J.E. (1983) *Algebraic Computing in General Relativity.* In the Proceedings of the 10th International Conference on General Relativity and Gravitation, Padua, July 3-8, edited by B. Bertotti, F. de Felice and A. Pascolini. pp. 139-162. Reidel Publishing.

[68] ▪ Cohen-Tannoudji C., Dui B. and Laloe F. (1978) *Quantum Mechanics* (2 volumes). John Wiley.

[69] ▪ Cohen-Tannoudji C., Dupont-Roc J. and Grynberg G. (1989) *Photons and Atoms: Introduction to Quantum Electrodynamics.* John Wiley.

[70] ▪ Cole C.A., Wolfram S., *et al.* (1981) *SMP Handbook.* Caltech.

[71] ▪ Collart S., Kalkbrener M. and Mall D. (1996) *Converting Bases with the Gröbner Walk.* Journal of Symbolic Computation **24**(4) pp. 465-469.

[72] ▪ Collins G.E. (1971) *The SAC-I System: An Introduction and Survey.* In the Proceedings of SYMSAM '71, the 2nd Symposium on Symbolic and Algebraic Manipulation, Los Angeles, March 23-25, edited by S.R. Petrick. pp. 144-152. ACM Press.

[73] ▪ Collins G.E. (1983) *Quantifier Elimination for Real Closed Fields: A Guide to the Literature.* In: Computer Algebra - Symbolic and Algebraic Computation, 2nd edition, edited by B. Buchberger, G.E. Collins and R. Loos, pp. 79-81. Springer-Verlag.

[74] ▪ Collins G.E. (1985) *The SAC-2 Computer Algebra System.* In the Proceedings, vol.2, of EUROCAL '85: European Conference on Computer Algebra, Linz, April 1-3, edited by B.F. Caviness. Lecture Notes in Computer Science **204**, pp. 34-35. Springer-Verlag.

[75] ▪ Condon E.U. and Shortley G.H. (1935) *The Theory of Atomic Spectra.* Cambridge University Press.

[76] ▪ Corbett R., Stallman R. and Hansen W. (1999) *Bison*, version 1.2.5, a computer program. Skunkware Development Tools, Santa Cruz Operation Inc. http://www.sco.com/skunkware/devtools/

[77] ▪ Courcelle B. (1990) *Graph Rewriting: An Algebraic and Logic Approach.* In: Handbook of Theoretical Computer Science, vol. 2, edited by J. van Leeuwen, pp. 193-242. MIT Press/Elsevier.

[78] ▪ Crampin M. and Pirani F.A.E. (1986) *Applicable Differential Geometry.* London Mathematical Society Lecture Notes Series, vol. 59. Cambridge University Press.

[79] ▪ Cyganowski S. and Carminati J. (1998) *The Maple Package NPTOOLS; a Symbolic Algebra Package for Tetrad Formalisms in General Relativity.* Computer Physics Communications **115** pp. 200-214.

[80] ▪ Czapor S.R. and McLenaghan R.G. (1987) *NP: A Maple Package for Performing Calculations in the Newman-Penrose Formalism.* General Relativity and Gravitation **19**(6) pp. 623-635.

[81] ▪ d'Inverno R. (1992) *Introducing Einstein's Relativity.* Oxford University Press.

[82] ▪ d'Inverno R.A. (1975) *Algebraic Computing in General Relativity*. General Relativity and Gravitation **6** pp. 567-593.

[83] ▪ d'Inverno R.A. (1980) *A Review of Algebraic Computing in General Relativity*. In: General Relativity and Gravitation: One Hundred Years after the Birth of Albert Einstein, edited by A. Held, pp. 491-537. Plenum Press.

[84] ▪ d'Inverno R.A. (1998) *Applications of SHEEP in General Relativity*. Computer Physics Communications **115** pp. 330-349.

[85] ▪ Dalmas S., Gaëtano M. and Watt S. (1997) *An OpenMath 1.0 Implementation*. In the Proceedings of ISSAC '97: International Symposium on Symbolic and Algebraic Computation, Kihei, Maui, Hawaii, July 21-23, edited by W.W. Küchlin. pp. 241-248. ACM Press.

[86] ▪ Davenport J.H. (1989) *Algebraic Computations and Structures*. In: Computer Algebra, edited by D.V. Chudnovsky and R.D. Jenks, Dekker.

[87] ▪ Davenport J.H. (1994) *Computer Algebra, Past, Present and Future*. pdf file (unpublished). University of Bath, http://www.bath.ac.uk/expertise/People/masjhd.html.

[88] ▪ Davenport J.H., Siret Y. and Tournier E. (1993) *Computer Algebra, Systems and Algorithms for Algebraic Computation*, 2nd edition. Academic Press.

[89] ▪ Davie A.J.T. (1992) *An Introduction to Functional Programming Systems Using Haskell*. Cambridge Computer Science Texts, vol. 27. Cambridge University Press.

[90] ▪ Dershowitz N. and Jouannaud J.-P. (1990) *Rewrite Systems*. In: Handbook of Theoretical Computer Science, vol. 2, edited by J. van Leeuwen, pp. 243-320. MIT Press/Elsevier.

[91] ▪ Di Blasio P. and Temperini M. (1993) *Subtyping Inheritance in Languages for Symbolic Computation Systems*. In the Proceedings of DISCO '93: Design and Implementation of Symbolic Computation Systems, Gmunden, Austria, Sept. 15-17, edited by A. Miola. Lecture Notes in Computer Science **722**, pp. 107-121. Springer-Verlag; also in J. Symbolic Computing 19 (1995) pp.39-63.

[92] ▪ Di Blasio P. and Temperini M. (1995) *Subtyping Inheritance and its Application in Languages for Symbolic Computation*. Journal of Symbolic Computation **19**(1-3) pp. 39-63.

[93] ▪ Dixon J.D. and Mortimer B. (1996) *Permutation Groups*. Graduate Texts in Mathematics. Springer-Verlag.

[94] ▪ Doleh Y. and Wang P.S. (1990) *SUI: A System Independent User Interface for an Integrated Scientific Computing Environment*. In the Proceedings of ISSAC '90: International Symposium on Symbolic and Algebraic Computation, Tokyo, Aug. 20-24, edited by S. Watanabe and M. Nagata. pp. 88-94. Addison-Wesley.

[95] ▪ Dolzmann A. and Sturm T. (1997) *REDLOG: Computer Algebra Meets a Computer Logic*. ACM SIGSAM Bulletin **31**(2) pp. 2-9.

[96] ▪ Doughty N.A. (1990) *Lagrangian Interaction- An Introduction to Relativistic Symmetry in Electrodynamics and Gravitation*. Addison Wesley.

[97] ▪ Dueck G. (1993) *New Optimization Heuristics: The Great Deluge Algorithm and the Record-to-Record Travel*. Journal of Computational Physics **104** pp. 86-92.

[98] ▪ Dueck G. and Scheuer T. (1990) *Threshold Accepting: A General Purpose Optimization Algorithm Appearing Superior to Simulated Annealing*. Journal of Computational Physics **90** pp. 161-175.

[99] ▪ Dunford N. and Schwartz J. (1988) *Linear Operators, Vols. 1-3*. John Wiley.

[100] ▪ Duval D. and Reynaud J.-C. (1994) *Sketches and Computation (Part I): Basic Definitions and Static Evaluation*. Mathematical Structures in Computer Science **4**(2) pp. 185-238.

[101] ▪ Duval D. and Reynaud J.-C. (1994) *Sketches and Computation (Part II): Dynamic Evaluation and Applications*. Mathematical Structures in Computer Science **4**(2) pp. 239-271.

[102] ▪ Edmonds A.R. (1996) *Angular Momentum in Quantum Mechanics*. Princeton University Press.

[103] ▪ Eiffel (1999) *http://www.cm.cf.ac.uk/CLE*.

[104] ▪ Enderton H. (1972) *A Mathematical Introduction to Logic*. Academic Press.

[105] ▪ Evans C.R. (1992) *Recent Progress in Computational Relativity*. In: Recent Advances in General Relativity, edited by A.I. Janis and J.R. Porter, pp. 260-264. Birkhäuser.

[106] ▪ Fano U. and Rau A.R. (1996) *Symmetries in Quantum Physics*. Academic Press.

[107] ▪ Feagin J.M. (1994) *Quantum Methods with Mathematica*. Telos/Springer-Verlag.

[108] ▪ Fernández F.M., Guardiola R. and Ros J. (1998) *Computer Algebra and Large Scale Perturbation Theory*. Computer Physics Communications **115** pp. 170-182.

[109] ▪ Fiedler B. (1997) *VECTAN (for vector analysis)*, version 1.1, a computer package for Mathematica. University of Leipzig.

[110] ▪ Fitch J., ed. (1992) *International Symposium DISCO'92*. Lecture Notes in Computer Science **721**. Springer-Verlag.

[111] ▪ Fleischer J. and Grabmeier J., eds. (1995) *Computer Algebra in Science and Engineering*. Bielefeld, Germany, 28-31 August.

[112] ▪ Fournier R., Kajler N. and Mourrain B. (1993) *IZIC: a Portable Language-driven Tool for Mathematical Surfaces Visualization*. In the Proceedings of DISCO'93: Design and Implementation of Symbolic Computation Systems, Gmunden, Austria, Sept. 15-17, edited by A. Miola. Lecture Notes in Computer Science **722**, pp. 341-353. Springer-Verlag.

[113] ▪ Fournier R., Kajler N. and Mourrain B. (1995) *Visualization of Mathematical Surfaces: the IZIC Server Approach*. Journal of Symbolic Computation **19**(1-3) pp. 159-173.

[114] ▪ Fraleigh J.B. (1998) *A First Course in Abstract Algebra*, 6th edition. Addison-Wesley.

[115] ▪ Frick I. (1977) *SHEEP*, a computer program for handling components of tensors. Dept. of Computer Science, Royal Institute of Technology, Stockholm.

[116] ▪ Frick I. (1986) *Algebraic Computing*. In the Proceedings of the 11th International Conference on General Relativity and Gravitation, Stockholm, July 6-12, edited by M.A.H. MacCallum. pp. 253-261. Cambridge University Press.

[117] ▪ Froberg R. (1997) *Introduction to Gröbner Bases*. John Wiley.

[118] ▪ Fuchssteiner B. (1996) *MuPad User's Manual - MuPad Version 1.2.2*. John Wiley.

[119] ▪ Fuchssteiner B., Wiwianka W., *et al.* (1993) *MuPAD Multi-Processing Algebra Data Tool*. Birkhäuser.

[120] ▪ Fulling S.A., King R.C., *et al.* (1992) *Normal Forms for Tensor Polynomials*. Classical and Quantum Gravity **9** pp. 1151-1197.

[121] ▪ GAP Group (1998) *GAP – Groups, Algorithms and Programming*, version 4.B.1, a computer program. Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, and School of Mathematical and Computational Sciences, University of St. Andrews, Scotland. http://www-groups.dcs.st-and.ac.uk./~gap/

[122] ▪ Garey M. and Johnson D. (1979) *Computers and Intractability (A Guide to the Theory of NP-Completeness)*. Bell Telephone Labs.

[123] ▪ Gaylord R.J., Kamin S.N., *et al.* (1995) *An Introduction to Programming with Mathematica*, 2nd edition. Springer-Verlag.

[124] ▪ Geddes K.O., Czapor S.R. and Labahn G. (1992) *Algorithms for Computer Algebra*. Kluwer Academic.

[125] ▪ Ghezzi C. and Jazayeri M. (1987) *Programming Language Concepts*, 2nd edition. John Wiley.

[126] ▪ Goguen J., Kirchner C., *et al.* (1987) *An Introduction to OBJ 3*. In 1st International Workshop, Conditional Term Rewriting Systems, Orsay, France, July 8-10, edited by S. Kaplan and J.-P. Jouannaud. Lecture Notes in Computer Science **308**, pp. 258-263. Springer-Verlag.

[127] ▪ Goguen J. and Malcolm G. (1997) *Algebraic Semantics of Imperatiave Programs*. MIT Press.

[128] ▪ Goguen J.A., Winkler T., *et al.* (2000) *Introducing OBJ*. In: Software Engineering with OBJ: Algebraic Specification in Action, edited by G. Malcolm, p. to appear. Kluwer.

[129] ▪ Goldberg A. and Robson D. (1983) *Smalltalk-80. The Language and its Implementation*. Addison-Wesley.

[130] ▪ Goldstein H. (1980) *Classical Mechanics*. Addison-Wesley.

[131] ▪ Grabmeier J., Kaltofen and Weispfenning, eds. *Computer Algebra Handbook*.

[132] ▪ Gray J.W. (1997) *Mastering Mathematica: Programming Methods and Applications*. Academic Press.

[133] ▪ Gray S., Kajler N. and Wang P.S. (1996) *Pluggability Issues in the Multi Protocol*. In the Proceedings of DISCO'96: Design and Implementation of Symbolic Computation Systems, Karlsruhe, Germany, 18-20 Sept., edited by J. Calmet and C. Limongelli. Lecture Notes in Computer Science **1128**, pp. 343-356. Springer-Verlag.

[134] ▪ Greif J.M. (1985) *The SMP Pattern Matcher*. In the Proceedings of EUROCAL '85: European Conference on Computer Algebra, Linz, edited by B.F. Caviness. vol. 2, Lecture Notes in Computer Science **204**, pp. 303-314. Springer-Verlag.

[135] ▪ Greiner W. (1990) *Relativistic Quantum Mechanics: Wave Equations*. Theoretical Physics, vol. 3. Springer-Verlag.

[136] ▪ Griesmer J.H. and Jenks R.D. (1971) *SCRATCHPAD/1---An Interactive Facility for Symbolic Mathematics*. In the Proceedings of the 2nd Symposium on Symbolic and Algebraic Manipulation. pp. 42-58. ACM Press.

[137] ▪ Griesmer J.H., Jenks R.D. and Yun D.Y.Y. (1975) *SCRATCHPAD User's Manual*. IBM Research, Yorktown Heights.

[138] ▪ Griffiths D.F. (1987) *Introduction to Elementary Particles*. John Wiley.

[139] ▪ Grivas G. and Maeder R.E. (1993) *Matching and Unification for the Object-oriented Symbolic Computation System AlgBench*. In the Proceedings of DISCO '93: Design and Implementation of Symbolic Computation Systems, Gmunden, Austria, Sept. 15-17, edited by A. Miola. Lecture Notes in Computer Science **722**, pp. 164-176. Springer-Verlag; also in J. Symbolic Computation 19 (1995), pp.39-63.

[140] ▪ Grivas G. and Palinginis A. (1996) *Compiling Residuation for a Multiparadigm Symbolic Programming Language*. In the Proceedings of DISCO'96: Design and Implementation of Symbolic Computation Systems, Karlsruhe, Germany, Sept. 18-20, edited by J. Calmet and C. Limongelli. Lecture Notes in Computer Science **1128**, Springer-Verlag.

[141] ▪ Grozin A.G. (1996) *Using REDUCE in High Energy Physics*. Cambridge University Press.

[142] ▪ Harbison S.P. (1992) *Modula-3*. Prentice Hall.

[143] ▪ Harlander R. and Steinhauser M. (1999) *Automatic Computation of Feynman Diagrams*. Progress in Particle and Nuclear Physics **43** pp. 167-228.

[144] ▪ Harper J.F. and Dyer C.C. (1994) *Tensor Algebra with REDTEN: A User Manual (a Reduce package)*, version 1.0. Department of Astronomy, Scarborough College, University of Toronto.

[145] ▪ Harper J.F. and Dyer C.C. (1999) *Redten*, version 4.1, a computer package for REDUCE. University of Toronto. http://www.scar.utoronto.ca/~harper/reten.html

[146] ▪ Harper R., McLean D. and Milner R. (1986) *Standard ML*. Report ECS-LFCS-86-2. Edinburgh University.

[147] ▪ Harris J.F. (1994) *Inheritance of Rewrite Rule Structures Applied to Symbolic Computation*. In the Proceedings of ISSAC'94: International Symposium on Symbolic and Algebraic Computation, Oxford, UK, 20-22 July. pp. 318-323. ACM Press.

[148] ▪ Harris J.F. (1999) *Semantica: Semantic Pattern Matching in Mathematica*. The Mathematica Journal (in press)

[149] ▪ Harrison R. (1993) *Abstract Data Types in Standard ML*. John Wiley.

[150] ▪ Hartley D. (1996) *Overview of Computer Algebra in Relativity*. In: Relataivity and Scientific Computing - Computer Algebra, Numerics, Visualization, edited by F.W. Hehl, R.A. Puntigam and H. Ruder, pp. 173-191. Springer-Verlag.

[151] ▪ Haskell Group (1998) *HUGS(Haskell User's Gofer System) 1.4*, version 98, a computer program.

[152] ▪ Heal K.M., Hansen M.L. and Rickard K.M. (1998) *Maple V Learning Guide (Version A)*, 2nd edition. Springer-Verlag.

[153] ▪ Hearn A.C. (1995) *DUMMY (for manipulating dummy indices)*, a computer package for REDUCE, 3.7. Konrad-Zuse-Zentrum, Berlin.

[154] ▪ Hearn A.C. (1995) *HEPHYS High Energy Physics*, a computer package for REDUCE, 3.6.

[155] ▪ Hearn A.C. (1998) *REDUCE User's and Contributed Packages Manual*, version 3.7. Konrad-Zuse-Zentrum, Berlin.

[156] ▪ Hearn A.C. and Fitch J.P. (1996) *REDUCE User's Manual*. RAND Publication CP78, Santa Monica, CA.

[157] ▪ Heck A. (1993) *FORM for Pedestrians*. CAN Expertise Centre.

[158] ▪ Heck A. (1996) *A Bird's-Eye View of Gröbner Bases*. CAN Expertise Centre.

[159] ▪ Heering J., Klint P. and Rekers J. (1990) *Incremental Generation of Parsers*. IEEE Transactions on Software Engineering **16**(12) pp. 1344-1350.

[160] ▪ Helton J.W., Miller R.L. and Stankus M. (1999) *NCAlgebra (for non-commutative algebra)*, a computer package for Mathematica, 3.0. MathSource 0208-998.

[161] ▪ Hibbard A. and Levvasseur K. (1998) *Exploring Abstract Algebra with Mathematica* (2 volumes). Springer-Verlag.

[162] ▪ Hoare C.A.R. (1973) *Hints on Programming Language Design (Keynote address)*. In Conference Record of POPL '97: The 1st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, MA, October. ACM Press.

[163] ▪ Horbatsch M. (1995) *Quantum Mechanics Using Maple*. Springer.

[164] ▪ Hörnfeldt L. (1979) *A System for Automatic Generation of Tensor Algorithms and Indicial Tensor Calculus*. In Symbolic and Algebraic Computation. Proceedings of EUROSAM '79, an International Symposium on Symbolic and Algebraic Manipulation, Marseille, France, June 26-28, edited by E.W. Ng. Lecture Notes in Computer Science **72**, pp. 279-290. Springer-Verlag.

[165] ▪ Hörnfeldt L. (1988) *STENSOR - User Guide and Reference Manual*. University of Stockholm, Institute of Theoretical Physics, Stockholm.

[166] ▪ Horspool R.N. (1990) *Incremental Generation of LR Parsers*. Computer Languages **15**(4) pp. 205-233.

[167] ▪ Horstmann C.S. and Cornell G. (1999) *Core Java 1.2*. Sun Microsystems Press.

[168] ▪ Hsieh A. and Yehudai E. (1992) *HIP: Symbolic High-Energy Physics Calculations*. Computers in Physics **6**(3) pp. 253-261.

[169] ▪ Hughes J. (1989) *Why Functional Programming Matters*. The Computer Journal **32**(2) pp. 98-107.

[170] ▪ Ilyin V., Kryukov A., *et al.* (1995) *CVIT: Dirac Gamma Matrices*, a computer package for REDUCE, 3.7.

[171] ▪ Ilyin V.A. and Kryukov A.P. (1991) *Symbolic Simplification of Tensor Expressions Using Symmetries, Dummy Indices, and Identities*. In the Proceedings of ISSAC '91: International Symposium of Symbolic and Algebraic Computation, Bonn, July 15-17, edited by S. Watt. pp. 224-228. ACM Press.

[172] ▪ Ilyin V.A. and Kryukov A.P. (1996) *ATENSOR - REDUCE Program for Tensor Simplification*. Computer Physics Communications **96** pp. 36-52.

[173] ▪ Itzykson C. and Zuber J.-B. (1980) *Quantum Field Theory*. McGraw-Hill.

[174] ▪ Jamin M. and Lautenbacher M.E. (1993) *Tracer Version 1.1: A Mathematica Package for γ-Algebra in Arbitrary Dimensions*. Computer Physics Communication **74** pp. 265-288.

[175] ▪ Jansson P. and Jeuring J. (1997) *PolyP - a Polytypic Programming Language Extension*. In Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, Jan. 15-17, edited by N.D. Jones. pp. 470-482. ACM.

[176] ▪ Jansson P. and Jeuring J. (1998) *PolyLib - a Library of Polytypic Functions*. In Proceedings, Workshop on Generic Programming, Chalmers University of Technology.

[177] ▪ Jenks R.D. and Sutor R. (1992) *AXIOM, the Scientific Computation System*. Springer-Verlag.

[178] ▪ Jeuring J. and Jansson P. (1996) *Polytypic Programming*. In Advanced Functional Programming, edited by J. Launchbury, E. Meijer and T. Sheard. Lecture Notes in Computer Science **1129**, pp. 68-114. Springer-Verlag.

[179] ▪ Judd B.R. (1975) *Angular momentum Theory for Diatomic Molecules*. Academic Press.

[180] ▪ Kadlecsik J. (1996) *Ricci Calculus Package in REDUCE*. Computer Physics Communications **93** pp. 265-282.

[181] ▪ Kajler N. (1990) *Building Graphic User Interfaces for Computer Algebra Systems*. In the Proceedings of DISCO '90: Design and Implementation of Symbolic Computation Systems, Capri, Italy, April 10-12, edited by A. Miola. Lecture Notes in Computer Science **429**, pp. 235-244. Springer-Verlag.

[182] ▪ Kajler N. (1992) *Building a Computer Algebra Environment by Composition of Collaborative Tools*. In the Proceedings of DISCO '92: Design and Implementation of Symbolic Computation Systems, Bath, UK, April 13-15, edited by J. Fitch. Lecture Notes in Computer Science **721**, pp. 85-94. Springer-Verlag.

[183] ▪ Kajler N. (1992) *CAS/PI: a Portable and Extensible Interface for Computer Algebra Systems*. In the Proceedings of ISSAC '92: International Symposium on Symbolic and Algebraic Computation, Berkeley, CA, July 26-29, edited by P.S. Wang. pp. 376-386. ACM Press.

[184] ▪ Kajler N. (1993) *User Interfaces for Symbolic Computation: A Case Study*. In the Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology, Atlanta, GA, USA. pp. 1-10. ACM Press.

[185] ▪ Kajler N. and Soiffer N. (1998) *A Survey of User Interfaces for Computer Algebra Systems*. Journal of Symbolic Computation **25**(2) pp. 127-159.

[186] ▪ Kaku M. (1993) *Quantum Field Theory: A Modern Introduction*. Oxford University Press.

[187] ▪ Kandri-Rodi A. and Weispfenning V. (1990) *Non-commutative Gröbner Bases in Algebras of Solvable Type*. Journal of Symbolic Computation **9**(1) pp. 1-26.

[188] ▪ Kavian M., McLenaghan R.G. and Geddes K.O. (1995) *MapleTensor: A new System for Performing Indicial and Component Tensor Calculations by Computer*. In The 6th Canadian Conference on General Relativity and Relativistic Astrophysics, Fredericton, NB, May 25-27, edited by S.P. Braham, J.D. Gegenberg and R.J. McKellar. pp. 269-272. American Mathematical Society.

[189] ▪ Kavian M., McLenaghan R.G. and Geddes K.O. (1996) *MapleTensor: Progress Report on a new System for Performing Indicial and Component Tensor Calculations Using Symbolic Computation.* In the Proceedings of ISSAC'96: International Symposium on Symbolic and Algebraic Computation, ETH, Zurich, 24-26 July, edited by Y.N. Lakshman. pp. 204-211. ACM Press.

[190] ▪ Keller B. and Struble C. (1999) *Opal (for computing Gröbner bases in non-commutative algebras)*, a computer package for REDUCE. University of Vermont. http://hal.cs.vt.edu/opal

[191] ▪ Kittel C. (1996) *Introduction to Solid State Physics*, 7th edition. John Wiley.

[192] ▪ Klioner S.A. (1994) *EinS (for calculations with indexed objects)*, a computer package for Mathematica, 3. Lohrmann Observatory, Dresden Technical University, Dresden.

[193] ▪ Klioner S.A. (1998) *New System for Indicial Computation and its Applications in Gravitational Physics.* Computer Physics Communications **115**(2-3) p 231–244.

[194] ▪ Knuth D.E. (1984) *The TeX book.* Addison-Wesley.

[195] ▪ Knuth D.E. (1991) *Semi-numerical Algorithms,* vol. 2, 2nd edition. The Art of Computer Programming. Addison-Wesley.

[196] ▪ Knuth D.E. (1992) *Literate Programming.* Report 27. Centre for the Study of Language and Information, Stanford University, Palo Alto, CA.

[197] ▪ Knuth D.E. (1998) *Sorting and Searching,* vol. 3, 2nd edition. The Art of Computer Programming. Addison-Wesley.

[198] ▪ Koonin S.E. and Meredith D.C. (1990) *Computational Physics*, 4th edition. Addison Wesley.

[199] ▪ Krasinski A. and Perkowski M. (1981) *ORTOCARTAN—A New Computer Program for Analytic Calculations in General Relativity.* General Relativity and Gravitation **13**(1) pp. 67-77.

[200] ▪ Krausz F. (1989) *A Better MACSYMA User Interface.* MACSYMA Newsletter **6**(3) pp. 10-13.

[201] ▪ Kredel H. and Pesch M. (1996) *MAS (Modula-2 Algebra System)*, version 1.00, a computer package. University of Passau, Germany. http://www.fmi.uni-passau.de/aalgebra/projects/mas.html

[202] ▪ Lee J.M. (1998) *Ricci: A Mathematica Package for Doing Tensor Calculations in Differential Geometry*, version 1.32, a computer package for Mathematica, 3.0 & 4.0. Department of Mathematics, University of Washington, Seattle, WA. http://www.math.washington.edu/~lee/Ricci

[203] ▪ Leler W. and Soiffer N. (1985) *An Interactive Graphical Interface for Reduce.* ACM SIGSAM Bulletin **19**(3) pp. 17-23.

[204] ▪ Limongelli C., Mele M.B., *et al.* (1990) *Abstract Specification of Mathematical Structures and Methods.* In the Proceedings of DISCO '90: Design and Implementation of Symbolic Computation Systems, Capri, Italy, April 10-12, edited by A. Miola. Lecture Notes in Computer Science **429**, pp. 61-70. Springer-Verlag; for extended version, cf. Theoretical Computer Science 104 (1992) pp.89-107.

[205] ▪ Limongelli C. and Temperini M. (1992) *Abstract Specification of Structures and Methods in Symbolic Mathematical Computation.* Theoretical Computer Science **104**(1) pp. 89-107.

[206] ▪ Lindgren I. and Morrison J. (1986) *Atomic Many-Body Theory,* vol. 3, 2nd edition. Springer-Verlag.

[207] ▪ Lippman S.B., Lajoie J. and Lajoie J. (1998) *C++ Primer*, 3rd edition. Addison-Wesley.

[208] ▪ Lucic V. (1995) *Dill: An Algorithm and a Symbolic Software Package for Doing Classical Supersymmetry Calculations.* Computer Physics Communications **92** pp. 90-110.

[209] ▪ MacCallum M.A.H. (1987) *Symbolic Computation in Relativity Theory.* In the Proceedings of EUROCAL '87: European Conference on Computer Algebra, Leipzig, June 2-5, edited by J.H. Davenport. Lecture Notes in Computer Science **378**, Springer-Verlag.

[210] ▪ MacCallum M.A.H. (1988) *Computer Algebra in Relativistic Gravity*. In the Proceedings of the 5th Marcel Grossmann Meeting on General Relativity, Perth, Australia, Aug. 8-13, edited by D.G. Blair and M.J. Buckingham. pp. 1157-1164. World Scientific, Singapore.

[211] ▪ MacCallum M.A.H. (1996) *Computer Algebra and Applications in Relativity and Gravity*. In Recent Developments in Gravitation and Mathematical Physics: Proceedings of the First Mexican School on Gravitation and Mathematical Physics, edited by A. Macias, T. Matos, O. Obregon*et al.* World Scietific Publishing.

[212] ▪ MacCallum M.A.H., Skea J.E.F., *et al.* (1994) *Algebraic Computing in General Relativity*. Clarendon Press.

[210] ▪ MacCallum M.A.H., Skea J.E.F., *et al.*, eds. (1989) *Algebraic Computing in General Relativity*, vol. 1. Lecture notes from the First Brazilian School on Computer Algebra. Oxford University Press.

[214] ▪ MacDonald N. (1994) *REDUCE for Physicists*. Institute of Physics Publishing.

[215] ▪ MacLane S. (1998) *Categories for the Working Mathematician*, 2nd edition. Graduate Texts in Mathematics **5**. Springer-Verlag.

[216] ▪ MacLane S. and Birkhoff G. (1979) *Algebra*, 2nd edition. Macmillan.

[217] ▪ Macsyma Corp. (1983) *MACSYMA Reference Manual*, version 10. Laboratory for Computer Science, MIT.

[218] ▪ Macsyma Group (1992) *ATENSOR (for exterior calculus of differential forms)*, a computer package for Macsyma.

[219] ▪ Macsyma Group (1992) *CTENSOR (for component tensor computation)*, a computer package for Macsyma.

[220] ▪ Macsyma Group (1992) *ITENSOR (for indicial tensor computation)*, a computer package for Macsyma.

[221] ▪ Maeder R. (1992) *AlgBench: An Object-oriented Symbolic Core System*. In the Proceedings of DISCO '92: Design and Implementation of Symbolic Computation Systems, Bath, UK, April 13-15, edited by J. Fitch. Lecture Notes in Computer Science **721**, pp. 56-64. Springer-Verlag.

[222] ▪ Maeder R.E. (1996) *Programming in Mathematica*, 3rd edition. Addison-Wesley.

[223] ▪ Magma Group (1998) *Magma*, version 2.4, a computer program. Computational Algebra Group, School of Mathematics and Statistics, University of Sydney, Australia. http://www.maths.usyd.edu.au:/u/magma/index.html

[224] ▪ Martin W.A. and Fateman R.J. (1971) *The MACSYMA System*. In the Proceedings of SYMSAM '71, the 2nd Symposium on Symbolic and Algebraic Manipulation, Los Angeles, March 23-25, edited by S.R. Petrick. pp. 59-75. ACM.

[225] ▪ MathSoft (1997) *MathCAD 7.0 User's Guide*. MathSoft Inc., Cambridge, MA.

[226] ▪ McCarthy J. (1950) *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I.* Communications of ACM **3**(4) pp. 184-195.

[227] ▪ McCarthy J., Abrahams P.W., *et al.* (1965) *LISP 1.5 Programmer's Manual, 2nd edition*. MIT.

[228] ▪ Melenk H. and Apel J. *NCPOLY (for non-commutative polynomial ideals, with the non-commutivity defined via Lie-bracket commutators)*, a computer package for REDUCE.

[229] ▪ Mendelson E. (1994) *Introduction to Mathematical Logic*, 3rd edition. van Nostrand.

[230] ▪ Mertig R. (1998) *FeynCalc 3.0 Reference Manual*. Mertig Research & Consulting, Amsterdam Science Park, Amsterdam, http://www.mertig.com.

[231] ▪ Mertig R., Böhm M. and Denner A. (1991) *Feyn Calc – Computer-algebraic Calculation of Feynman Amplitudes*. Computer Physics Communications **64** pp. 345-359.

[232] ▪ Merzbacher E. (1997) *Quantum Mechanics*, 3rd edition. John Wiley.

[233] ▪ Messiah A. (1981) *Quantum Mechanics, Vol.1-2*. North Holland.

[234] ▪ Mével A., Guéguen T. and Wolczko M. (1987) *Smalltalk-80*. Macmillan Computer Science Series. Basingstoke: Macmillan Education.

[235] ▪ Meyer B. (1992) *Eiffel : The Language*. Prentice Hall Object-oriented Series. Prentice Hall.

[236] ▪ Meyer B. (1997) *Object-Oriented Software Construction*, 2nd edition. Prentice Hall.

[237] ▪ Milner R., Tofte M., *et al.* (1997) *The Definition of Standard ML*, Revised edition. MIT Press.

[238] ▪ Miola A., ed. (1990) *International Symposium DISCO'90*. Lecture Notes in Computer Science **429**. Springer-Verlag.

[239] ▪ Miola A., ed. (1993) *International Symposium DISCO'92*. Lecture Notes in Computer Science **722**. Springer-Verlag.

[240] ▪ Misner C.W., Thorne K.S. and Wheeler J.A. (1973) *Gravitation*. Freeman.

[241] ▪ Missura S.A. (1993) *Extending AlgBench with a Type System*. In the Proceedings, of DISCO '93: Design and Implementation of Symbolic Computation Systems, Gmunden, Austria, Sept. 15-17, edited by A. Miola. Lecture Notes in Computer Science **722**, pp. 359-363. Springer-Verlag.

[242] ▪ Modula-3 (1999) *http://www.research.digital.com/SRC/modula-3/html/home.html*.

[243] ▪ Monagan M.B. (1993) *Gauss: a Parameterized Domain of Computation System with Support for Signature Functions*. In the Proceedings of DISCO '93: Design and Implementation of Symbolic Computation Systems, Gmunden, Austria, Sept. 15-17, edited by A. Miola. Lecture Notes in Computer Science **722**, pp. 81-94. Springer-Verlag.

[244] ▪ Monagan M.B., Geddes K.O., *et al.* (1998) *Maple V Programming Guide (Version A)*. Springer-Verlag.

[245] ▪ Monk J.D. (1980) *Introduction to Set Theory*. Krieger Publications.

[246] ▪ Mora F. (1985) *Gröbner Bases for Non-commutative Polynomial Rings*. In Proceedings of AAECC-3, Algebraic Algorithms and Error-Correcting Codes, Grenoble, July, edited by J. Calmet. Lecture Notes in Computer Science **229**, pp. 353-362. Springer-Verlag.

[247] ▪ Mora T. (1994) *An Introduction to Commutative and Non-commutative Gröbner Bases*. Theoretical Computer Science **134** pp. N1:131-173.

[248] ▪ Musgrave P., Pollney D. and Lake K. (1995) *GRTesnorII: A Computer Algebra System for General Relativity*. In The 6th Canadian Conference on General Relativity and Relativistic Astrophysics, Fredericton, NB, May 25-27. p. 313–318.

[249] ▪ Musgrave P., Pollney D. and Lake K. (1998) *GRTensorII*, version 1.7 for Maple (GRTensorII); 1.2 for Mathematica (GRTensorM), a computer program. Department of Physics, Queen's University, Kingston, Ontario.

[250] ▪ NCAlgebra (1995) *NCAlgebra*, a computer package for Mathematica, 2.2 or later.

[251] ▪ Nguyen N.-A. and Nguyen-Dang T.T. (1998) *Symbolic Calculations of Unitary Transformations in Quantum Dynamics*. Computer Physics Communications **115** p 183–199.

[252] ▪ Noble J., Taivalsaari A. and Moore I., eds. (1999) *Prototype-based Programming: Concepts, Languages, and Applications*. Springer-Verlag.

[253] ▪ Norman A. and Fitch J. (1996) *Interfacing REDUCE to Java*. In the Proceedings of DISCO'96: Design and Implementation of Symbolic Computation Systems, Karlsruhe, Germany, Sept. 18-20, edited by J. Calmet and C. Limongelli. Lecture notes in Computer Science **1128**, pp. 271-276. Springer-Verlag.

[254] ▪ OBJ (1999) *http://www-cse.ucsd.edu/users/goguen/sys/obj.html*.

[255] ▪ Ohanian H.C. and Rufkia R. (1994) *Gravitation and Spacetime*, 2nd edition. W. W. Norton.

[256] ▪ Open Source Software (1999) *Flex*, version 2.5.4, a computer program. Skunkware Development Tools, Santa Cruz Operation Inc. http://www.sco.com/skunkware/devtools/

[257] ▪ Pang T. (1997) *Computational Physics*. Cambridge University Press.

[258] ▪ Parker L. and Christensen S.M. (1994) *MathTensor: A System for Doing Tensor Analysis by Computer*. Addison-Wesley.

[259] ▪ Paulson L.C. (1996) *ML for the Working Programmer*, 2nd edition. Cambridge University Press.

[260] ▪ Peltio M. (1999) *Summa.m*, a computer package for Mathematica. MathSource 0210-508.

[261] ▪ Petitot M. (1993) *Experience with Axiom*. In New Trends and Developments. Proceedings of SC'93, the International IMACS Symposium on Symbolic Computation, Lille, France, June 14-17, edited by G. Jacob, N.E. Oussous and S. Steinberg. pp. 240-240. LIFL University Lille.

[262] ▪ Petkovsek M., Wilf H. and Zeilberger D. (1996) $A = B$. A.K. Peters Ltd.

[263] ▪ Pitt D., Aabramsky S., *et al.*, eds. (1985) *Category Theory and Computer Programming, Tutorial and Workshop, Guildford, U.K., Sept. 16-20*. Lecture Notes in Computer Science **240**. Springer-Verlag.

[264] ▪ Plasmeijer R. and van Eekelen M. (1993) *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley.

[265] ▪ Plasmeijer R. and van Eekelen M. (1995) *Concurrent Clean Language Report*, version 1.0.

[266] ▪ Portugal R. (1998) *An Algorithm to Simplify Tensor Expressions*. Computer Physics Communications **115**(2-3) pp. 215-230.

[267] ▪ Portugal R. and Sautú S.L. (1997) *Applications of Maple to General Relativity*. Computer Physics Communications **105** pp. 233-253.

[268] ▪ Postel F. and Hillebrand R. (1999) *Flexible Mathematical GUI Controls for Mathematical Electronical Documents*. MathPad **9**(1) pp. 10-17.

[269] ▪ Press W.H., Teukolsky S.A., *et al.* (1993) *Numerical Recipes in C: The Art of Scientific Computing*, 2nd edition. Cambridge University Press.

[270] ▪ Qin H. (1997) *Symbolic Vector Analysis Using Mathematica*. In 1997 Mathematica Developer Conference, Champaign, Illinois. http://library.wolfram.com/conferences/devconf97/

[271] ▪ Qin H., Tang W.M. and Rewoldt G. (1999) *Symbolic Vector Analysis in Plasma Physics*. Computer Physics Communications **116** pp. 107-120.

[272] ▪ Racah G. (1942) *Theory of Complex Spectra I*. Physical Review **61** pp. 186-197.

[273] ▪ Racah G. (1942) *Theory of Complex Spectra II*. Physical Review **62** pp. 438-462.

[274] ▪ Racah G. (1943) *Theory of Complex Spectra III*. Physical Review **63** pp. 367-382.

[275] ▪ Redfern D. and Campbell C. (1998) *The MATLAB 5 Handbook*. Springer-Verlag.

[276] ▪ Rose J.S. (1978) *A Course on Group Theory*. Cambridge University Press.

[277] ▪ Rotenberg M. (1959) *3j and 6j Symbols*. Technology Press, MIT.

[278] ▪ Ryder L.H. (1985) *Quantum Field Theory*. Cambridge University Press.

[279] ▪ Sage M.L. (1988) *An Algebraic Treatment of Quantum Vibrations*. Journal of Symbolic Computation **5**(3) pp. 377-384.

[280] ▪ Sakurai J.J. and Tuan S.F. (1985) *Modern Quantum Mechanics*. Benjamin/Cummings.

[281] ▪ Santas P.S. (1993) *A Type System for Computer Algebra*. In the Proceedings of DISCO '93: Design and Implementation of Symbolic Computation Systems, Gmunden, Austria, Sept. 15-17, edited by A. Miola. Lecture Notes in Computer Science **722**, pp. 177-191. Springer-Verlag; also in J. Sym. Comput. 19 (1995) #1-3, pp.79-109.

[282] ▪ Santas P.S. (1995) *A Type System for Computer Algebra*. Journal of Symbolic Compuation **19**(1-3) pp. 79-109.

[283] ▪ Sarkar A. (1996) *Incremental Parser Generation for Tree Adjoining Grammars*. Report CL/9809029. University of Pennsylvania, Philadelphia.

[284] ▪ Savage E. (1999) *Higher Symmetries in Jahn-Teller Systems*. Ph.D. Thesis, Dept. of Physics, University of Canterbury.

[285] ▪ Scandron M. (1979) *Advanced Quantum Theory and its Applications through Feynman Diagrams*. Springer-Verlag.

[286] ▪ Schrüfer E. (1987) *EXCALC User's Manual*. RAND Corp., Santa Monica.

[287] ▪ Schrüfer E. (1995) *EXCALC (for calculations in the calculus of modern differential geometry)*, a computer package for REDUCE, 3.7.

[288] ▪ Schrüfer E., Hehl F.W. and McCrea J.D. (1987) *Exterior Calculus on the Computer: The REDUCE-Package EXCALC Applied to General Relataivity and to the Poincaré Gauge Theory*. General Relativity and Gravitation **19**(2) pp. 197-218.

[289] ▪ SciFace Software (1999) *MuPad*, version 1.4, a computer program. SciFace Software GmbH&Co. KG, Germany, http://www.sciface.com.

[290] ▪ Sebesta R.W. (1998) *Concepts of Programming Languages*, 4th edition. Addison-Wesley.

[291] ▪ Semenov A. (1998) *LanHEP - a Package for Automatic Generation of Feynman Rules from the Lagrangian*. Computer Physics Communications **115** p 124–139.

[292] ▪ Shankar R. (1980) *Principles of Quantum Mechanics*. Plenum Press.

[293] ▪ Shaw W.T. and Tigg J. (1994) *Applied Mathematica: Getting Started, Getting it Done*. Addison-Wesley.

[294] ▪ Sherrying J. and Prince G. (1993) *Dimsym (Symmetry Determination and Linear Differential Equations Package)*, a computer package for REDUCE. School of Mathematics, La Trobe University, Melbourne. http://www.latrobe.edu.au/www/mathstats/Maths/Dimsym

[295] ▪ Simmonds J.G. (1994) *A Brief on Tensor Analysis*, 2nd edition. Undergraduate Texts in Mathematics. Springer-Verlag.

[296] ▪ Sims C. (1970) *Determing the Conjugacy Classes of a Permutation Group*. In Computers in Algebra and Number Theory, edited by G. Brikhoff and M. Hall. vol. 4, SIAM-AMS American Mathematics Society.

[297] ▪ Sims C.C. (1971) *Computation with Permutation Groups*. In the Proceedings of SYMSAM '71, the 2nd Symposium on Symbolic and Algebraic Manipulation, Los Angeles, March 23-25, edited by S.R. Petrick. pp. 23-28. ACM Press.

[298] ▪ Skea J.E.F. (1988) *RSHEEP: A Combination of the Algebra Systems SHEEP and REDUCE*. In the Proceedings of the 5th Marcel Grossmann Meeting on General Relativity, Perth, Australia, Aug. 8-13, edited by D.G. Blair and M.J. Buckingham. vol. 2, p. 1165–1168. World Scientific.

[299] ▪ Skiena S.S. (1990) *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Addison Wesley.

[300] ▪ Skiena S.S. (1998) *The Algorithm Design Manual*. TELOS.

[301] ▪ Smith R.B. and Ungar D. (1996) *Programming as an Experience: The Inspiration for Self*. Sun Microsystems Laboratories.

[302] ▪ Soft Warehouse Inc. (1991) *The DERIVE User Manual: a Mathematical Assistant for your Personal Computer*, version 2. Soft Warehouse Inc., Honolulu, HI.

[303] ▪ Soleng H.H. (1995) *CARTAN (for tensor analysis)*, version 1.2, a computer package for Mathematica. Scandinavian University Press, Oslo, Norway.

[304] ▪ Soleng H.H. (1996) *Tensors in Physics: The CARTAN User's Guide and Reference Manual*. Scandinavian University Press, Oslo.

[305] ▪ Soleng H.H. (1997) *Tensors in Physics: Doing Black Holes and Cosmology on a PC.* In 1997 Mathematica Developer Conference, Champaign, Illinois. http://library.wolfram.com/conferences/devconf97/

[306] ▪ Stedman G.E. (1989) *Diagram Techniques in Group Theory.* Cambridge University Press.

[307] ▪ Stroustrup B. (1997) *The C++ Programming Language*, 3rd edition. Addison-Wesley.

[308] ▪ Strubbe H. (1974) *Manual for SCHOONSHIP, a CDC 6000/7000 Program for Symbolic Evaluation of Algebraic Expressions.* Computer Physics Communications **8**(??) p 1–30.

[309] ▪ Takayama N. (1998) *KAN (for computations in the rings of polynomials, differential operators, difference operators and q-difference operators)*, version 2.981129, a computer program. Kobe University. http://www.math.kobe-u.ac.jp/KAN

[310] ▪ Thompson S. (1999) *Haskell : The Craft of Functional Programming*, 2nd edition. International Computer Science Series. Addison-Wesley.

[311] ▪ Thompson W.J. (1994) *Angular Momentum: An Illustrated Guide to Rotational Symmetries for Physical Systems.* John Wiley.

[312] ▪ Trott M. (1999) *The Mathematica Guidebook: Concepts, Examples and Applications.* Springer-Verlag.

[313] ▪ Ufnarovski V. (1998) *Introduction to Noncommutative Gröbner Bases Theory.* In Gröbner Bases and Applications, edited by B. Buchberger and F. Winkler. London Mathematical Society Lecture Note Series **251**, pp. 259-280. Cambridge university Press.

[314] ▪ Ullman J.D. (1997) *Elements of ML Programming : ML97*, ML97 edition. Prentice Hall.

[315] ▪ Ungar D. (1991) *The Power of Simplicity.* Lisp and Symbolic Computation **4**(3) pp. 187-205.

[316] ▪ Unicode-Consortium (1999) *The Unicode Standard, Version 2.0.* Addison-Wesley Longman.

[317] ▪ van den Bergh N. (1988) *ORTHOFRAME: a MAPLE Package for Performing Calculations in the Orthonormal Tetrad Formalism.* Classical and Quantum Gravity **5**(10) p L169–L179.

[318] ▪ Van der Linden P. (1999) *Just Java 2.* Prentice Hall Java Series. Sunsoft: Prentice Hall.

[319] ▪ Veltman M. (1991) *Schoonschip*, a computer program for PowerMac. Ann Arbor, MI.

[320] ▪ Vermaseren J.A.M. (1990) *FORM, a Program for the Manipulation of Very Large Formulae.* In: Computer Algebra in Physical Research, vol. 2, p. 35–42. World Scientific.

[321] ▪ Vermaseren J.A.M. (1992) *Symbolic Manipulation with FORM version 2: Tutorial and Reference Manual.* CAN, Amsterdam.

[322] ▪ Vermaseren J.A.M. (1993) *FORM*, version 2.3, a computer program. CAN, Amsterdam, The Netherlands.

[323] ▪ Villegas R. (1999) *Working with Unevaluated Expressions.* In 1999 Mathematica Developer's Conference, Champaign, Illinos, October 21-23. http://library.wolfram.com/conferences/devconf99/

[324] ▪ Vlasov A.Y. (1999) *Object-oriented and Functional Programming for Symbolic Manipulation.* Report SC/ 9901006. Federal Centre for Radiology, St. Petersburg, Russia.

[325] ▪ Wagner D.B. (1996) *Power Programming with Mathematica: The Kernel.* McGraw-Hill.

[326] ▪ Wald R.M. (1984) *General Relativity.* University of Chicago Press.

[327] ▪ Walters R.F.C. (1992) *Categories and Computer Science.* Cambridge Computer Science Texts **28**. Cambridge University Press.

[328] ▪ Waterloo Maple (1990) *CLIFFORD*, a computer package for Maple. MapleSoft, Waterloo, Ontario.

[329] ▪ Waterloo Maple (1997) *Maple*, version V release 4, a computer program. Maplesoft, Waterloo, Ontario. http://www.maplesoft.com

[330] ▪ Watson D. (1989) *High-level Languages and their Compilers.* Addison-Wesley.

[331] ▪ Watt S.M., Broadbery P.A., *et al.* (1994) *A First Report on the A# Compiler*. In the Proceedings of ISSAC '94: International Symposium on Symbolic and Algebraic Computation, Oxford, UK, 20-22 July, edited by J.v.z. Gathen and M. Giesbrecht. pp. 25-31. ACM Press.

[332] ▪ Weber A. (1993) *On Coherence in Computer Algebra*. In the Proceedings of DISCO'93: Design and Implementation of Symbolic Computation Systems, Gmunden, Austria, Sept. 15-17, edited by A. Miola. Lecture Notes in Computer Science **722**, pp. 95-106. Springer-Verlag.

[333] ▪ Weinberg S. (1972) *Gravitation and Cosmology*. John Wiley.

[334] ▪ Weissbluth M. (1978) *Atoms and Molecules*. Academic Press.

[335] ▪ Wick G.C. (1950) Phys. Rev. **80** p 268.

[336] ▪ Wielandt H. (1964) *Finite Permutation Groups*. Academic Press.

[337] ▪ Winkler F. (1996) *Polynomial Algorithms in Computer Algebra*. Texts & Monographs in Symbolic Computation. Springer-Verlag.

[338] ▪ Winston P.H. and Horn B.K. (1984) *LISP*, 2nd edition. Addison-Wesley.

[339] ▪ Wirsing M. (1990) *Algebraic Specification*. In: Handbook of Theoretical Computer Science, vol. 2, edited by J. van Leeuwen, pp. 675-788. MIT Press/Elsevier.

[340] ▪ Wirth N. (1974) *On the Design of Programming Languages*. In the Proceedings of the IFIP Congress, Stockholm. pp. 386-393.

[341] ▪ Wolfram S. (1988) *Mathematica, A System for Doing Mathematics by Computer*, 2nd edition. Addison Wesley.

[342] ▪ Wolfram S. (1996) *The Mathematica Book*, 3rd edition. Wolfram Media/Cambridge University Press.

[343] ▪ Wolfram S. (1999) *The Mathematica Book 4.0*, 4th edition. Wolfram Media/Cambridge University Press.

[344] ▪ Wybourne B.G. (1970) *Symmetry Principles and Atomic Spectroscopy*. Wiley-Interscience.

[345] ▪ Wybourne B. (1999) *Schur*, version 5.2, a computer program. http://smc.vnet/Schur.html

[346] ▪ Zeilberger D. (1998) *WZ Theory, Chapter II*. Report Temple University, Math.CO/9811070. Lecture given at MSRI Workshop on Symbolic Computation in Geometry and Analysis.

[347] ▪ Zhytnikov V.V. (1999) *GRG (for differential geometry, gravitation and field theory)*, a computer package for REDUCE, 3.2. Moscow State Pedagogical University.

[348] ▪ Zimmerman R.L. and Olness F.I. (1995) *Mathematica for Physics*. Addison-Wesley.